A Structural Operational Semantics for PLEX

Johan Erikson
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
johan.erikson@mdh.se

Abstract

Programming Language for EXchanges, PLEX, is a pseudo-parallel and event-driven real-time language developed by *Ericsson*. The language is designed for, and used in, central parts of the AXE telephone switching system. The language has a signal paradigm as its top execution level, and it is event-based in the sense that only events, encoded as signals, can trigger code execution. Due to the fact that a PLEX program file consist of several independent subprograms, in combination with an execution model where new jobs are spawned and put in queues, we also classify the language as pseudo-parallel. This report presents a *structural operational semantics* for fundamental parts of the language, i.e., over jumps and signal sending statements, and it should be seen in a further perspective, where the aim is to extend and modify the language with a possibility to run in a multi-processor environment.

Earlier attempts to map the language to description languages, like SDL, have not been as successful as expected, which is probably due to the fact that the semantics of the language **and** its execution model have not been paid enough attention. With this report, a formal basis for further investigations in that direction is provided.

Contents

List of Figures iv

List of Tables vii

1 Introduction 1

1.1 Background and Problem Definition 1
1.2 Limitations 2
1.3 Aim 2
1.4 Organization 2

I The Execution Model of APZ/PLEX 4

2 The AXE System and the PLEX Language 5

2.1 The AXE System 5
2.1.1 Central- and Regional Processors 5
2.1.2 The Application Modularity (AM) Concept 7
2.1.3 Input and Output statements 9
2.1.4 Load, Reload and Dump 10
2.2 Programming Language for EXchanges 11
2.2.1 The structure of a PLEX program 12
2.2.2 Records, Files and Pointers 14
2.2.3 Variables 15
2.2.4 Data Encapsulation 18

3 The Execution Model 20

3.0.5 PLEX structure and OS requirements 20
3.0.6 Software Units 20
3.0.7 Function Blocks 22

3.0.8 Application System 22
3.1 Program Interwork - Signals 22
3.1.1 Direct and buffered signals 25
3.1.2 Unique and multiple signals 25
3.1.3 Single and combined signals 26
3.1.4 Local and Non-local signals 28
3.1.5 Signals and Priorities 28
3.1.6 Signals and Data 28
3.2 Jobs, Signal Buffers and Job Handling 28
3.2.1 What is a Job? 29
3.2.2 Signal Buffers 29
3.2.3 Job Handling 31
3.2.4 Execution Time Limits 35
3.3 Linking Encapsulation 35
3.3.1 Addressing a Program Sequence 36
3.4 Software Recovery 39
3.4.1 Forlopp 41
3.4.2 System Restart 41
3.4.3 Forlopp Release or a System Restart? 43
3.4.4 Variables and Software Recovery 45

II Semantics **46**

4 Related Work **47**

5 Programming Language Semantics **49**
5.1 The meaning of a program 49
5.2 Semantic approaches 50
5.3 Notation 51
5.4 Operational Semantics 52
5.4.1 Natural Semantics 52
5.4.2 Structural Operational Semantics 53
5.5 Denotational Semantics 54
5.6 Axiomatic Semantics 55

6 Semantic Approach **57**
6.1 Selected Approach and Motivation 57
6.2 The State of the System 58
6.3 Lexical Units and Syntactical Categories 63
6.4 Statements for Variable Assignments 67
6.5 Jump Statements 67
6.6 Conditional Statements 68
6.7 Selections 69
6.8 Iterations 69
6.9 Signal Sending/Receiving Statements 70
6.9.1 Statements for Single Signals 71
6.9.2 Statements for Combined Signals 72
6.9.3 Statements for Local Signals 73
6.10 Exit 73
6.11 Semantic Functions 73
6.12 The Semantics for Assignment Statements 76
6.13 The semantics for Jump statements 79
6.14 The semantics for Conditional statements 79
6.15 The semantics for Selection statements 80
6.16 The semantics for Iteration statements 80
6.17 The Semantics for Signal Statements 83
6.17.1 Single Signals 86
6.17.2 Combined Signals 89
6.17.3 The Semantics for Local Signals 92
6.18 The Semantics for the EXIT-statement 93
6.19 The Semantic Function $\mathcal{S}_{PLEX}$ 95

7 Summary **97**

Index **100**

A The Semantics for PLEX **101**

B The Signal Description **109**

List of Figures

1.1 *This report and its context.* 3

2.1 *The (original) hierarchical structure of the AXE system. (The parts that will be of interest in this report is marked with bold text.)* 6
2.2 *Stores in the central processor (CP). (The interaction between the different stores are covered in Section 3.3.)* 7
2.3 *The AM concept incorporated into the AXE system.* 8
2.4 *The I/O system and its communication with the environment.* 10
2.5 *The different languages used in different parts of the AXE system* 13
2.6 *Structure of the SPI, i.e., a PLEX program file.* 15
2.7 *An example file with n records and a pointer with the current value 2.* 16
2.8 *Variables and properties (from a storage point of view).* . . 17
2.9 *Permitted combinations of variable properties and variable types.* 19
2.10 *The structure of a software unit (block). The possibility of several sub-programs accessing the same data within the block is shown. All sub-programs (signal entries) can access all DS variables inside the same block (except for individuals that are DS variables inside a record). This conveys a DS variable can be used as a communication channel between all sub-programs inside the same software unit.* 19

3.1 *APT Application system.* 21
3.2 *The different types of software signals.* 23

3.3 *A PLEX program file divided in subprograms. Note that the assignment `CUSELESS = 0;` will never be executed since it is placed between an exit and an enter statement. (See also Fig. 2.6 where a complete program file is described.)* 24
3.4 *Direct and buffered signals.* 25
3.5 *Unique and multiple signals.* 26
3.6 *Single and combined signals.* 27
3.7 *Possible properties for CP-CP signals. X indicates a legal/possible combination, shaded with Grey indicates an illegal alternative.* **NOTE**: *A combined* **backward** *signal can not be multiple since this signal is an answer (i.e., an acknowledgment) to a "caller" and must therefore return to the "caller" and nobody else.* 27
3.8 *Forward and Backward signals.* 27
3.9 *Sending of a delayed (and multiple) signal. The signal is sent from Unit A and received in Unit C but, as could be seen in the figure, it is possible to receive the signal in Unit B as well* ***if*** *Unit B is specified as the receiver by the PLEX designer.* 31
3.10 *Job buffers and runtime priorities in the AXE system.* 32
3.11 *The execution model - Four jobs are executed. The process of transfering a buffered signal from the sending block to the receiving, via a job buffer, is shown in Fig. 3.12. NOTE the "parallel" architecture that could become real parallel execution.* 33
3.12 *Linking and execution for a buffered signal in APZ. See also Fig. 3.11.* **NOTE**: *The procedure is the same for direct signals except that they* ***not*** *are inserted in a Job Buffer.* . . . 34
3.13 *PS, showing SDT, SST and Program Code of one function unit.* 36
3.14 *RS, showing the Reference Table.* 37
3.15 *The information flow in determining the signal entry when sending a signal.* 37
3.16 *The consecutive order of handling a signal sending.* 38
3.17 *Show how addressing to DS is performed in RS. BSA points to the starting point of BAT* 40

3.18 *Different types of system restart.* 43
3.19 *The intensity counter.* 44
3.20 *Different levels of recovery after a detected software error.* . 44
3.21 *Data security of different start/restart types.* 45

5.1 *The compound statement expressed in natural semantics.* . 53
5.2 *The compound statement expressed in structural operational semantics.* 54
5.3 *The compound statement expressed in denotational semantics.* 55
5.4 *The compound statement expressed in a axiomatic semantics style.* 56

6.1 *The different stores in the central processor (CP).* 59
6.2 *A simplified figure of the job buffers in the PLEX/AXE environment. (See also Fig. 3.10)* 60
6.3 *The FIFO-semantics of the job buffers.* 60
6.4 *Organization of the job buffers. NOTE: The pointer registers, W8-W9, are treated as one register (and referred to as POINTER B0). This is due to the fact that the register in W9 is only used when a pointer is too large to fit only in W8. In this case, the pointer will be split up and placed in W8 with its first half, and in W9 with its second half.* 61
6.5 *PLEX iteration statements - a comparison.* 70
6.6 *The process of finding the receiver of a buffered signal. (Repeated from Section 3.2.3, Fig. 3.12.)* **NOTE**: *The process is similar for direct signals, except that they are not inserted in a Job Buffer.* (See also Figure 6.4 for a detailed description on the organization of the job buffers.) 85

List of Tables

6.1 *Priorities and meaning of the PLEX operators. Priorities are numbered from the highest (1) to the lowest (8).* 66

6.2 *A summary of the semantic functions defined and used in Chapter 6 (and in Appendix A).* 74

6.3 *The semantics of ”arithmetic” expressions.* 76

6.4 *The semantics of ”boolean” expressions. (See also, Table 6.1 where the different operators, as well as their meaning, are described.)* a_n *represents a field-expression whereas* t_n *represents a string.* 77

6.5 *The function* $\mathcal{APZ}$ *which fetches the first ”ready-job” with highest priority.* 94

Chapter 1

Introduction

1.1 Background and Problem Definition

The programming language PLEX (Programming Language for EXchanges) is a pseudo-parallel and event-based real-time language developed by *Ericsson*. The language is designed for telephony systems and used in central parts of the AXE switching system (from Ericsson). The language has a signal paradigm as its top execution level, and it is event-based in the sense that only events, encoded as signals, can trigger code execution. The term pseudo-parallel has arisen due to the fact that a PLEX program file consist of independent sub-programs (which will be discussed in Section 3.1, and Fig. 3.3), in combination with an execution model (Fig. 3.11) where new jobs are spawned and put in different queues, called job buffers, for later execution.

The language has been continuously evolving since the 1970's when it was originally designed. But in parallel with this, attempts have been made to introduce more "modern" languages like *C++* for instance, or specification languages like *standard SDL*, and let the system execute this code as well as PLEX code. However, most of these attempts have failed and considerable money has been spent. It is probable that the failure is due to properties of the language and its execution model, i.e. the semantics of the language has not been paid enough attention. For example, to use SDL successfully, the unique semantics of PLEX were considered in creating an extension to SDL, called SDL-10 which could then be code generated to PLEX.

Until now, the semantics for PLEX has been defined through its im-

plementation. A formal semantics for the language can serve as an exact documentation, which could be referred to when the implementation is updated (e.g., when a new hardware platform is introduced).

This report should also be seen in a further perspective, where the aim is to extend and modify the language with a possibility to run in a multi-processor environment, see Fig. 1.1. This could already be done today due to the modular structure of the language, but as shown by Lindell [Lin03], there are problems that need to be solved.

1.2 Limitations

This report will focus on the basic concepts of signals since this is considered to be the most important aspects of the language, and the parts that most significantly differentiate PLEX from other languages. It will give an operational semantics for the individual PLEX statements, as well as for sequences of statements. However, the semantics for sequences of statements is restricted to *well-formed* constructs (which will be defined in Section 6.19).

1.3 Aim

The aim of this report is to give a semantic definition of the most important parts of PLEX, mentioned in Section 1.2. With this in hand, a formal basis for further investigations and comparisons with other languages is provided since the meaning of a (PLEX)program has been given a semantic definition. The semantics will also reveal ambiguities and prevent "ad hoc" solutions when the language is moved to a new hardware platform.

A second aim is to form the basis for further investigations in the direction of executing PLEX in a multi-processor environment.

1.4 Organization

This report is structured in two main parts:

- **Part I** includes the Technical Report "*The Execution Model of APZ/PLEX - An Informal Description*" by J. Erikson and B Lin-

dell [EL02]. This part serves as an introduction to the execution model of PLEX for the reader not familiar with the language and its environment[1]. *This part could be skipped, without any loss, by the reader familiar with PLEX.*

- **Part II** is the main part of this report. This part deals with the semantics for PLEX. Chapter 5 serves as an introduction to semantic notation and describes the most common frameworks. Chapter 6 could be seen as the *main chapter* of this report, since the semantics for PLEX is defined here. The semantics is developed throughout the chapter, when we look at the different statements in the language. Appendix A then summarizes the semantics that are defined in Chapter 6.

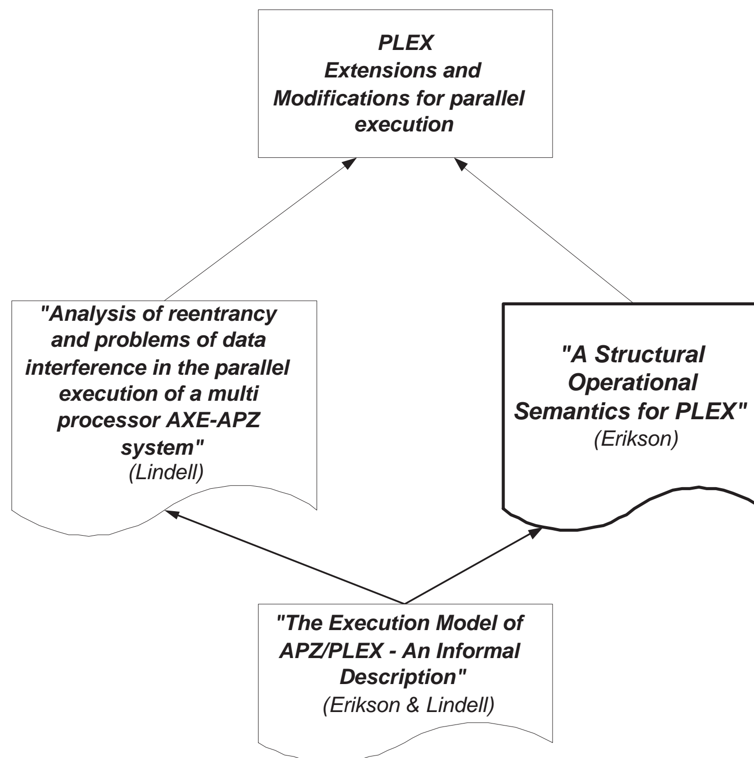


Figure 1.1: *This report and its context.*

[1]Also, a general survey of PLEX as well as of the AXE switching system can be found in [AGG99]

Part I

The Execution Model of APZ/PLEX

Chapter 2

The AXE System and the PLEX Language

2.1 The AXE System

The AXE telephone exchange system from Ericsson, developed in its earliest version in the beginning of the 1970s, is structured in a modular and hierarchical way. It consists of the two main parts:

APT: The telephony or switching part

APZ: The control part including central and regional processors

which both consist of hardware **and** software. The two main parts are divided into subsystems.

A subsystem is divided in function blocks. Function blocks consist of function units which is either a central software unit or a hardware unit, a regional software unit and a central software unit. The original structure of the system is shown in Fig 2.1.

Somewhere around 1994-95, the concept of *Application Modularity* (AM) was integrated into the system. This will be discussed in Section 2.1.2

2.1.1 Central- and Regional Processors

The hardware aspects that is of interest in this report is the distinction between Central- and Regional Processors. This is because different

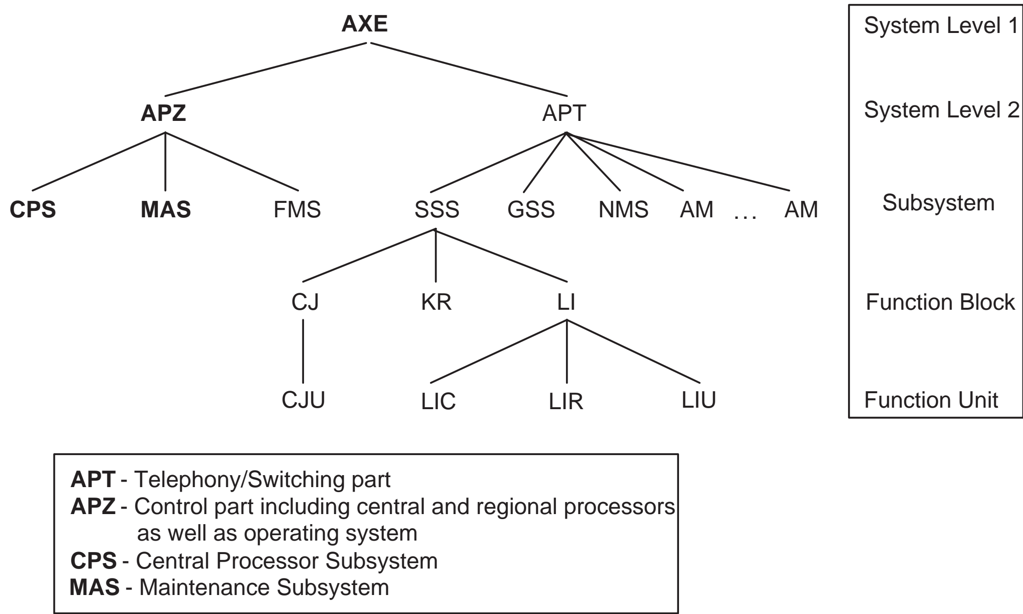


Figure 2.1: *The (original) hierarchical structure of the AXE system. (The parts that will be of interest in this report is marked with bold text.)*

forms of interwork is performed between different kinds of processors. The distinctions are briefly discussed in this subsection and explained in more detail in Section 3.1.

Regional Processor (RP): There are several regional processors in an AXE system. The main task of a regional processor is to relieve the central processor by handling small routine jobs like scanning and filtering.

Central Processor (CP): This is the central control unit of the system. All complex and non-trivial decisions are taken in the central processor. This is the place for all forms of non-routine work. The work of the processor can be separated into two specifically distinct parts, namely instruction execution and job administration. Instruction execution means handling of uninterrupted sequences of operations where the work consists of address table look-up and calculations, plausibility checks, storage accesses and data manipulations. The job administration mainly consists of signal handling, signal conversion and signal buffer handling. The execution

of instructions is a single-stream work by nature, whereas the job administration to a great extent is a question of prioritized job queues (Section 3.2) and transfer of signal data.

The CP is always duplicated. The two sides work in parallel, performing exactly the same operations. During normal operation, one CP is executive and the other is stand-by. A continuous check is made to ensure that both processors reach the same result - If they don't, some form of recovery action is performed (Section 3.4). The CP duplication also enables function changes (installation of new software versions) while the exchange is in an operational mode by first installing new software on the stand-by side and then change the executive and stand-by order between the processors. As a last step, the new software is installed on the former executive (now stand-by) side.

The CPs store all central software and data. The CP memory consists of the register memory and the different stores. Programs are stored in the program store (PS) and data is stored in the data store (DS). The reference store contains information about where to find the different programs and data, Fig. 2.2.

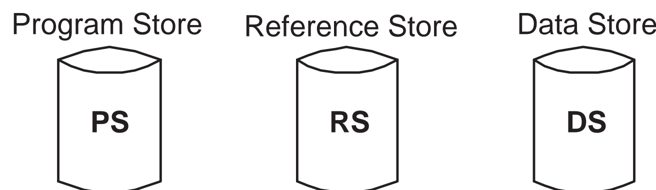


Figure 2.2: *Stores in the central processor (CP). (The interaction between the different stores are covered in Section 3.3.)*

2.1.2 The Application Modularity (AM) Concept

The AXE *Source System* is a number of hardware **and** software resources developed to perform specific functions according to the customer's requirements. It can be thought of as a ”basket” containing all the functionality available in the AXE system. Over the years, new

source systems has been developed by adding, updating or deleting functions in the original source system. But in the 1980's, the development of the AXE system for different markets (US, UK, Sweden, Asia, etc.) has led to parallel development of the source system since functionality could not easily be ported between different markets.

The solution to this increasing divergence was the *Application Modularity* (AM) concept, which made fast adaption to customer requirements possible. The AM concept specifically targeted the following requirements:

- the ability to freely combine applications in the system,
- quick implementation of requirements, and
- the reuse of existing equipment.

The basic idea is to gather related pieces of software (and hardware) into something called Application Modules (AMs). Different telecom applications, such as ISDN, PSTN (fixed telephony), and PLMN (Public Land Mobile Network), are then constructed by combining the necessary AMs. The idea is described in Fig. 2.3, where it is also shown that different AMs can be used in more than one application.

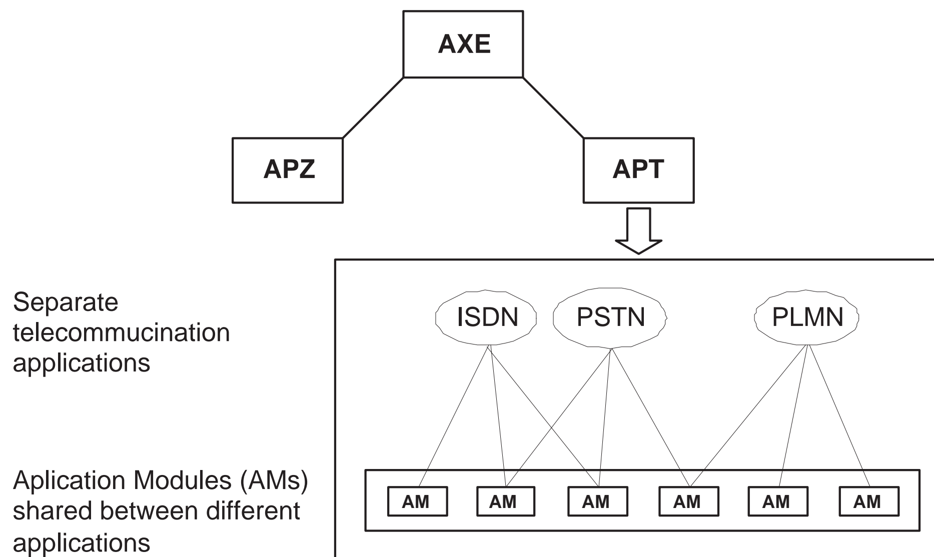


Figure 2.3: *The AM concept incorporated into the AXE system.*

The introduction of the AM concept ended the problem with parallel development of different source systems. Instead, with AMs as building blocks, the required exchange was constructed by combining the necessary AMs into an exchange with the required functionality (i.e., with the necessary applications).

2.1.3 Input and Output statements

An AXE exchange needs to communicate with its environment and its operation and maintenance (O&M) staff. Some typical situations could be the following:
- An exchange technician changes subscriber categories, replaces devices or connects new subscribers.
- The exchange informs the O&M staff of important events, e.g., if an RP is blocked due to a fault. In other words, the I/O statements are an important part of the recovery mechanism. (See Section 3.4.)
- Input/output includes certain routine tasks to, e.g. dumping data on a hard disk.

There is a large number of I/O devices used; alarm and hard copy printers, display units, work stations and PC's, magnetic tape drivers, hard and flexible disks.

Before communicating with an I/O device, the PLEX program has to seize the device. Likewise, the device has to be released when the communication ends. This guarantees exclusive access to the device. All I/O devices are connected to a support processor (SP), and function blocks that receive or send information via the I/O system are called **user blocks**. Fig. 2.4 shows the interaction between the I/O system and a user block. When seizing an I/O device, the I/O system assigns a free line buffer and a free analysis buffer (see Fig. 2.4) to this device. These buffers temporarily store the I/O text. The analysis buffer handles input from the I/O device, and the line buffer handles output.

The basic (PLEX) statements for transferring information between the buffers and the I/O device, and between the buffers and the user blocks are:
- `FETCH`: transfer information from the analysis buffer to the user block.
- `INSERT`: transfer information from the user block to the line buffer.
- `WRITE`: orders the I/O system to print out the text in the line buffer to

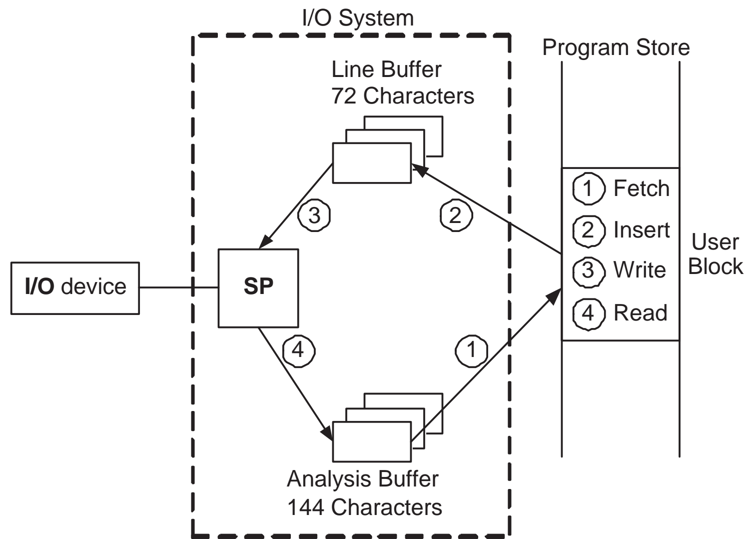


Figure 2.4: *The I/O system and its communication with the environment.*

an I/O device.
- `READ`: transfer information from the I/O device to the analysis buffer. Again, see Fig. 2.4.

Typically, I/O communication starts with the operator entering a command on an I/O device. The command is received by the I/O system and delivered to the software unit where it has been defined by the programmer. A command is received in a program (i.e., a software unit) in the same way as a signal (Section 3.1) but the command receiving statement must be preceded by the keyword **`COMMAND`** to indicate that this is a statement used by the I/O system.

2.1.4 Load, Reload and Dump

An AXE exchange may exist for up to 40 years, which implies certain requirements regarding the operation and maintenance of the software. The terms **Load**, **Reload** and **Dump** are covered in this section since they will be used in this report when we discuss variables (Section 2.2.3) and software recovery (Section 3.4).

When all the software blocks have been written and compiled, the programs and data, initial and exchange, are written, *dumped*, to a

magnetic tape which is loaded into the exchange. This process is called ***initial loading***. On loading of new blocks, or new revisions of existing blocks, an incremental re-linking occurs, as well as an initialization of data store variable values, if required according to their given *variable properties*[1]. A DCI (Data Conversion Information) is written for each block being loaded to specify the data initialization between the old (if existing) and new blocks. During the *function change process* (Section 2.1.1) the new block can get its new value from either of the following three ways:
- Get value from *data sector*[2].
- Get value from DCI.
- Get value from existing software.

In the case of system failure where a *system restart*[3] has been performed, software backup copies are *reloaded* into the exchange. When reloaded, some variables will receive reload values from the magnetic tape, whereas other variables will not have values until the program is executed by a *signal*[4]. Whether or not a variable receives a reload value is determined by the variable properties set by the designer. This is covered in Section 2.2.3.
Reloading means that the contents of DS (i.e., only RELOAD declared variables) are reloaded into the exchange again. If a change has occurred in PS and RS, they will be reloaded as well.

The contents of Program-, Reference- and Data store are regularly saved to a hard disk (or a magnetic tape). This process is called *dump* and enables the reload action described above.

2.2 Programming Language for EXchanges

Programming Language for EXchanges (PLEX) is designed by Ericsson and used to program telephony systems. It lacks common statements from other programming languages such as `WHILE` loops, negative numeric values and real numbers. These are not needed in a telephony

[1]Variable properties is covered in Section 2.2.3
[2]The data sector is mentioned in Section 2.2.1
[3]The system restart process is explained in Section 3.4
[4]Signals are examined in Section 3.1

exchange system. The language was designed and developed in its first form in the 1970s and extended in 1983. The version under consideration in this report, PLEX-C, is used in the AXE central processors (see Section 2.1.1). Other languages used in the AXE system are shown in Fig. 2.5[5]. The reason for developing a new language for the AXE system was that no other languages under consideration fulfilled Ericsson's requirements.

Some important characteristics of the language are listed below:

- PLEX is an event-based language with a signaling paradigm as the top execution level. Only events can trigger code execution and events are programmed as signals. A typical event is when a subscriber lifts the phone to dial a number.
  The execution model is described in Chapter 3 and signals in Section 3.1.

- The signals are executed on one of four priority levels (explained in Section 3.2), which results in very little overhead when a higher level interrupts a lower since each priority level has its own register set.

- Jobs (Section 3.2.1) at the same level are "atomic" and can never interrupt each other.

2.2.1 The structure of a PLEX program

When we talk about a PLEX program, or a PLEX program file, we mean the PLEX file that specifies a function unit (Section 3.0.7). This document, the *Source Program Information* (SPI), shown in Fig. 2.6, consists of the following main parts:

- The **Declare** sector, which contains the variable and constant declarations that are used in the program sector. Variables with the property `DS`, Data Store, (Section 2.2.3) will exist beyond the execution of subprograms.

[5] As could be seen in Fig. 2.5, there is another dialect of PLEX (PLEX-M). However, these dialects are similar, and when we talk about PLEX in this report, we mean the dialect used in the central processors, i.e, the PLEX-C dialect.

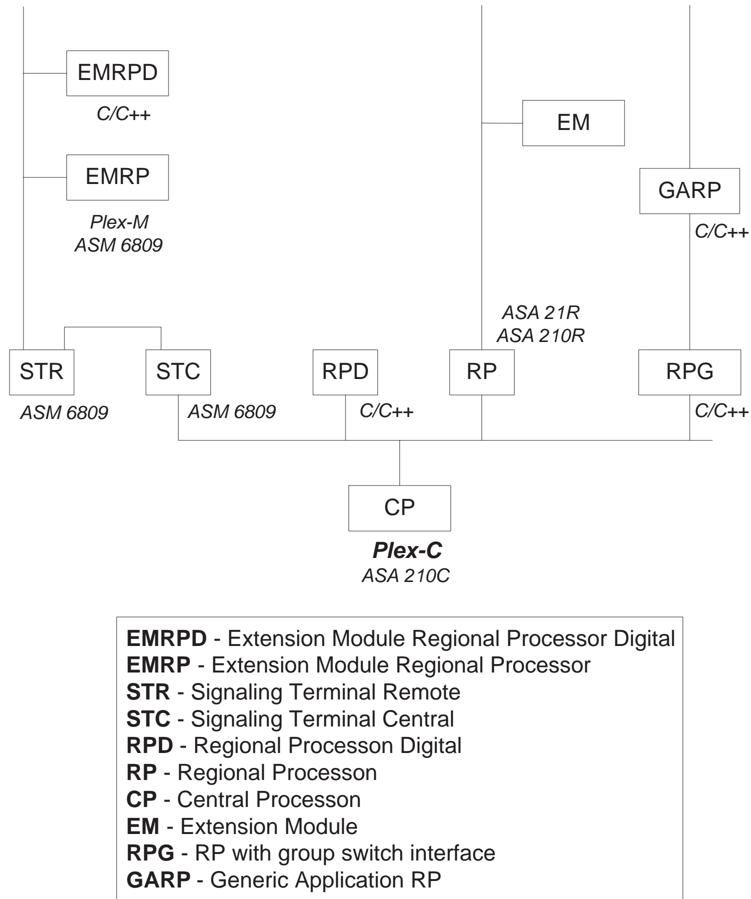


Figure 2.5: *The different languages used in different parts of the AXE system*

- The **Parameter** sector, where specific AXE parameters are placed. These parameters are not local to a block, and permit global access from all parts of the exchange. They can be *changed by customers* since they are placed in an SQL database.

- The **Program** sector contains the executable statements, i.e., the PLEX source code that will run in the exchange. This sector is normally divided in several subprograms (explained in Section 3.1 and Fig. 3.3).

- The **Data** sector: Some variables, i.e. Data Store variables, needs to have initial values when the program (i.e., the SPI) is loaded into the exchange[6]. These initial values can be provided in the data sector. Also, the position, i.e. the base address, of stored variables in memory can be allocated in the data sector. This enables a faster function change (briefly described in Section 2.1.1).

- The **ID** sector is used for internal documentation only.

The SPI is compiled together with the following documents[7]:
- The *Signal Survey*, SS, which is a list of all the different signals that one function unit (i.e., the function unit specified in the SPI) receives and sends. There is one SS per function unit. There is no information about senders and receivers in the SS, this information is added later during loading.
- The *Signal Description*, SD. The function blocks and function units communicate with signals (Section 3.1). The SD describes the purpose, type and data of *one* signal. SDs are stored in separate signal handling libraries.

2.2.2 Records, Files and Pointers

Records collect variables that describe properties of a group of items, for instance, calls or subscribers[8]. Record variables may be stored field, symbol or string variables (Section 2.2.3). Variables in a record may

[6]The *initial loading* is described in Section 2.1.4.

[7]The different steps of the compilation process, as well as the PLEX compiler, is described in [AE00]

[8]A (PLEX) record is similar to a `struct` in C.

```
DOCUMENT KRUPROGRAM;
DECLARE;
:
:
END DECLARE;
PARAMETER;
:
:
END PARAMETER;
PROGRAM; PLEX;
:
:
END PROGRAM;
DATA;
:
:
END DATA;
END DOCUMENT;
ID KRUPROGRAM TYPE DOCUMENT;
:
:
END ID;
```

Figure 2.6: *Structure of the SPI, i.e., a PLEX program file.*

be indexed or structured, and they are called individual variables. DS (Data Store, described in Section 2.2.3) variables that are not part of a record, are known as *common* variables.

A **File** is a set of records. One file consist of one or more records, all with the same individual variables.

Pointers address the relevant record in a file. In PLEX, pointers are simply record numbers. The records in a file are numbered, and the value of the pointer is the number of the current record. In other words, pointers in PLEX are **not** similar to pointers in C and can not be manipulated in the same way. Fig. 2.7 shows an example file with its records and a pointer. The number of records in a file may be fixed or changeable. A fixed size is specified in the Data sector of the SPI (Section 2.2.1), while alterable file sizes are set by commands (Section 2.1.3).

2.2.3 Variables

Depending on how variables is to be treated at a software error and a following recovery action, the PLEX designer can assign different properties to the variables. This is to be covered in this section.

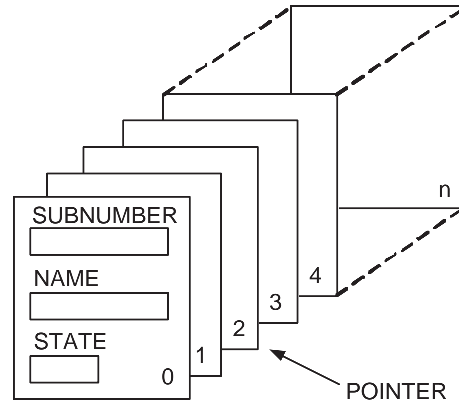


Figure 2.7: *An example file with n records and a pointer with the current value 2.*

There are three different data types in PLEX:
- *Field variables* for numeric information. They contain non-negative integers only. (Negative integers are not needed in the AXE system.)
- *Symbol variables* for symbol information, e.g., `IDLE, BLOCKED, BUSY`, etc.
- *String variables* store text strings.
These data types (variables) can be *stored* or *temporary*.

- The value of a temporary variable exists only in the Register Memory (RM - internal CP registers) and only while its corresponding software is being executed. Variables are by default temporary.

- Stored variables are stored in the Data Store (Fig. 2.2), loaded into a register in the RM for processing and then written back to the DS. Thus, its value is never lost, even if the program is exited and re-entered later. DS variables are also a natural way to communicate between different *forlopps*[9].

It is the stored variables that may be assigned the different properties already described. These properties are `DS, CLEAR, RELOAD, DUMP, STATIC, BUFFER` and `COMMUNICATION BUFFER`. The properties will all be described in this section.

[9]Forlopps are explained in Section 3.4.1

From a storage point of view, the variables can be divided into the following types: Temporary and stored have been described above. The third category is the buffers. Buffer variables[10] are allocated dynamically in an area reserved for dynamic buffers by using an allocate statement. The size of the buffers can be specified static (`COMMUNICATION BUFFER`) or dynamic `BUFFER`. The fixed size is specified in the Declare sector (Section 2.2.1) while the dynamic size can be set in the Program sector. The dynamic buffers are slower than the static since they must be administered dynamically. These categories are pictured in Fig. 2.8 together with its properties.

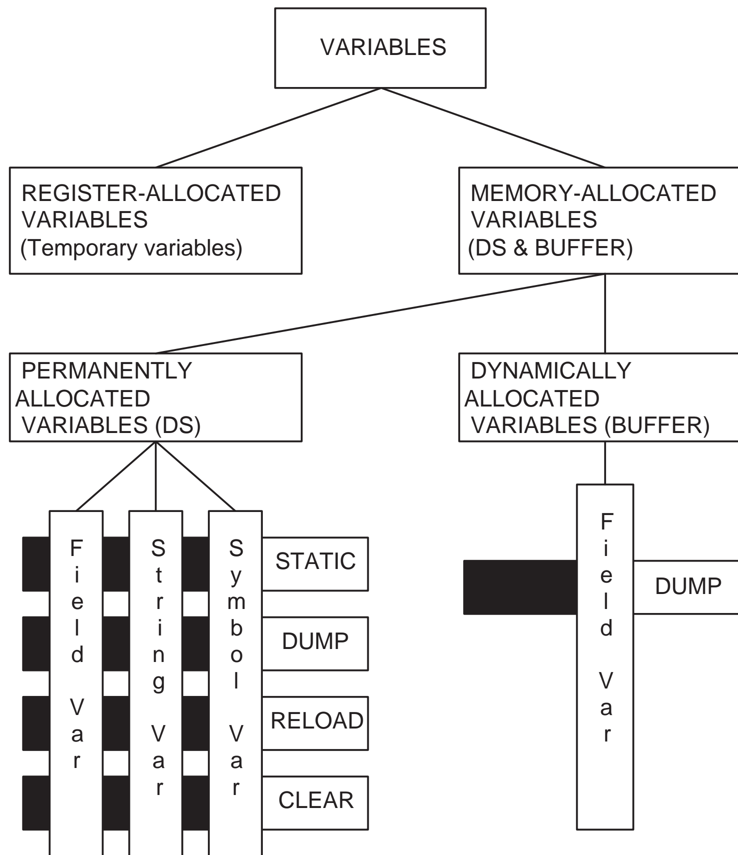


Figure 2.8: *Variables and properties (from a storage point of view).*

Under normal circumstances, the exchange starts the (application) software and it never stops. After serious errors, however, the APZ (i.e.,

[10]Buffer variables are similar to the `array` structure in C.

the operating system part) stops the program execution and restarts the software. The following properties describe the variable behavior at start or restart:

- `CLEAR` - "Clearing at start/restart"
  Field variables are set to zero; symbol variables to the first value in their declaration list.

- `RELOAD` - Loading at "restart with reload"
  The variable value is reloaded from tape/hard disk to ensure that the values before and after the "restart with reload" are the same.

- `DUMP` - "Dumping at restart".
  This property is used for testing and tracing purposes.

- `STATIC` - When a software unit in an operating exchange is to be updated, a *function change* takes place. Remember from Section 2.1.1 that the CP is always duplicated. This means that new software can be installed while the exchange is running. A `STATIC` declared variable means that the variable value is not updated with a new software version.

Not all combinations of the variable properties are possible (i.e., legal). Fig. 2.9 contains a table listing all valid combinations of variables and properties.

2.2.4 Data Encapsulation

All variables and constants declared in the *Declare* sector of the SPI, see Section 2.2.1, have their scope inside the software unit specified. All subprograms (Section 3.1) of that SPI can access these variables and constants. Subprograms not part of that function unit cannot access these variables and constants.

<table>
<tr><th></th><th>Field Variable</th><th>Symbol Variable</th><th>String Variable</th></tr>
<tr><td>DS
DS DUMP
DS STATIC</td><td colspan="3">Yes</td></tr>
<tr><td>DS RELOAD
DS RELOAD DUMP
DS RELOAD STATIC</td><td colspan="3">Yes</td></tr>
<tr><td>DS CLEAR
DS CLEAR DUMP</td><td colspan="2">Yes</td><td>No</td></tr>
<tr><td>BUFFER
BUFFER DUMP</td><td>Yes(1)</td><td colspan="2">No</td></tr>
<tr><td>Temporary</td><td colspan="2">Yes</td><td>No</td></tr>
</table>

(1) Except for one- and two-dimensional arrays

Figure 2.9: *Permitted combinations of variable properties and variable types.*

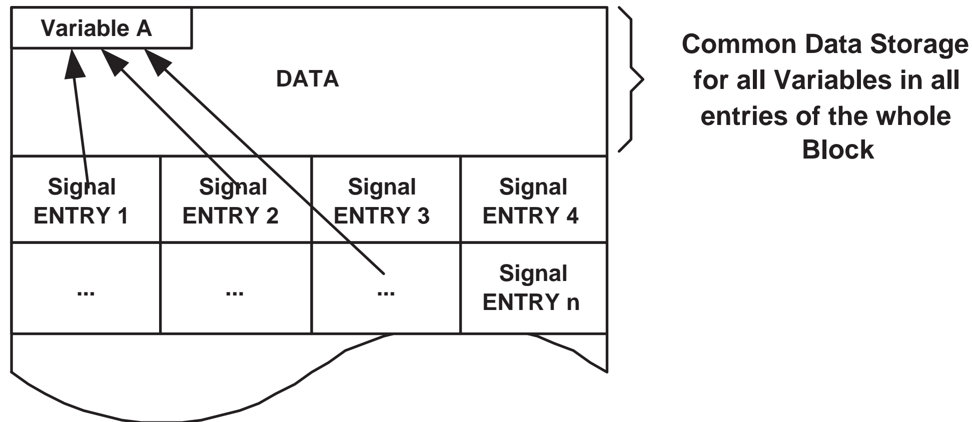


Figure 2.10: *The structure of a software unit (block). The possibility of several sub-programs accessing the same data within the block is shown. All sub-programs (signal entries) can access all DS variables inside the same block (except for individuals that are DS variables inside a record). This conveys a DS variable can be used as a communication channel between all sub-programs inside the same software unit.*

Chapter 3

The Execution Model

A brief discussion of the execution model has already been given in Section 2.2 and we continue and deepen the discussion in this section. We first briefly discuss PLEX structure, operating system requirements, function blocks and application system before we look deeper at program interwork (i.e, *signals*), Section 3.1, and job buffers, Section 3.2, both central concepts in the PLEX/APZ environment.

3.0.5 PLEX structure and OS requirements

PLEX is an asynchronous concurrent event based real-time language and, as stated in Section 2.2, it has a signaling paradigm as the top execution level which means that only events can trigger code execution and these events are programmed as signals. Signals will be further explored in Section 3.1. The main task of an operating system that is to run PLEX, is to buffer incoming signals and start their execution in the right signal entry statement.

3.0.6 Software Units

In large software systems, such as a telecommunication system, there is a need to group code into modules, for example, to control a certain hardware, or to implement in software add-on functionality. A Software Unit is a quantity of PLEX code for the different jobs[1] needed for such a module, called a function. A Unit can not access data in another unit,

[1]Jobs are covered in Section 3.2.1.

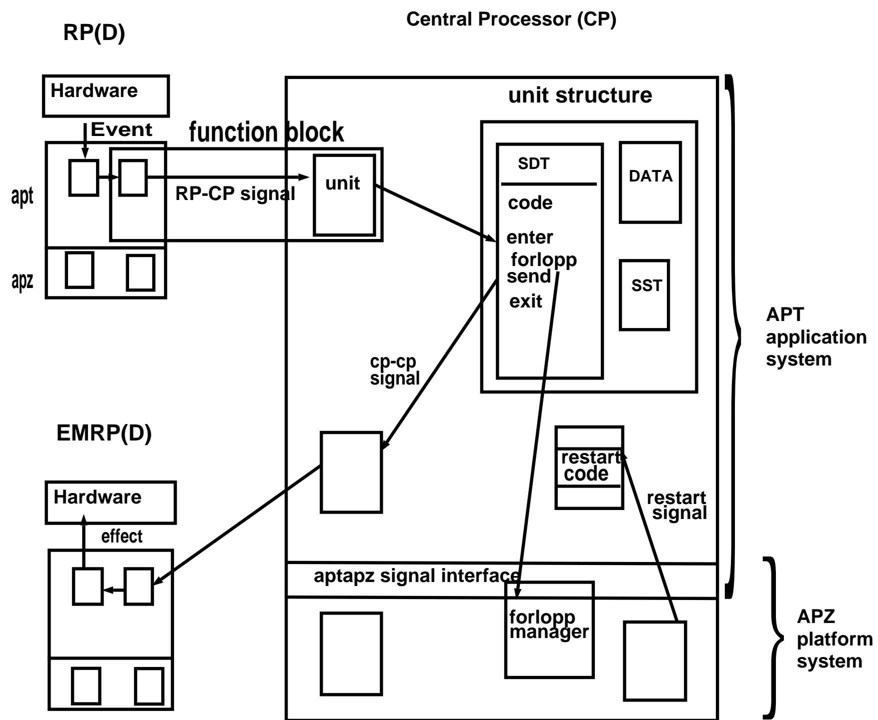


Figure 3.1: *APT Application system.*

i.e, a unit has data encapsulation (see Section 2.2.4).

3.0.7 Function Blocks

A function block is a software unit by itself or a software unit in the CP with the associated software unit in the EMRP or RP and possibly associated hardware needed to implement a function.

If we relate the function blocks to the AM concept, described in Section 2.1.2, it should be pointed out that an AM is not a PLEX language construct. From a PLEX language point of view, each AM and the common resources can be seen as a collection of blocks. Signals between AMs and to/from the common resources are gathered into standard interfaces.

3.0.8 Application System

An application system is a group of function blocks that interwork together to form a complete application, such as the control of a certain telephone exchange, see Fig. 3.1. All the signals and units of the part of the application system hosted on a certain processor take part in a "linking" process. (For units written in PLEX-C, the host is the CP.) The linking process resolves that signals sent from a certain unit are directed to the right entry point in the right unit.

3.1 Program Interwork - Signals

A signal is an externally defined language element in PLEX for the interwork between software units. A signal can be described as a message within one or between two software units or as an asynchronous (one way) function call, i.e., it is signals that perform the communication between different function units. Signals can be classified in numerous ways (Section 3.1.1, 3.1.2, 3.1.3 and 3.1.4) but the **main distinction** is between *direct* and *buffered* signals (Section 3.1.1). A direct signal is similar to a jump from one function unit or program to another, whereas a buffered signal is more like a `fork`[2] system call **except** that the ex-

[2] `fork` is a nonANSI C function that "*copies the current process and begins executing it concurrently*", [KP96]. The execution will then continue in this newly created "child-

ecution continues in the "*parent* process" whereas the "*child* process" is put in the job queue (Section 3.2) for later execution. In this way, after the sending of the buffered signal, the two execution paths are independent parallel threads, unsynchronized with each other. The difference is explained in more detail in Section 3.1.1, but we already state that buffered signals is the "norm" and that the classification referred to only applies to CP-CP signals. CP-RP and RP-CP signals are **always** buffered.

As shown in Fig. 3.2, signals are sent between software executing on the different processor types described in Section 2.1.1.

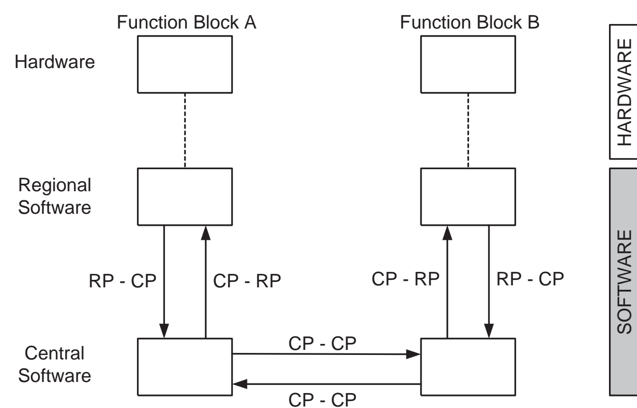


Figure 3.2: *The different types of software signals.*

Most signals could be seen as a jump from a signal-sending statement in one program to a signal-receiving statement in another program (even if buffered signals first go through a buffer). This implies that the code in a PLEX program unit[3] never executes from the beginning to the end (i.e., from the beginning of the program file to the end of the program file), but from a signal receiving statement (e.g., `ENTER`), to either a direct signal-sending statement (e.g., `SEND`) or an `EXIT` statement. In PLEX, a ***subprogram*** is the code sequence from `ENTER` to `EXIT`. It is possible to leave a subprogram with an `EXIT` without a previous signal sending statement, but it is also possible to send several buffered signals before an `EXIT` statement. Fig. 3.3 illustrates a gen-

process".

[3]A PLEX program unit = a PLEX source code file

eral program divided into subprograms. Note that since programs written in PLEX do not normally execute from start to end, or in any order, it can not be assumed that the program in Fig. 3.3 receives SIGNAL1 before or after SIGNAL3, or SIGNAL4 before or after SIGNAL6. This can result in unpredictable values of stored variables.

```
PROGRAM; PLEX;
  ENTER SIGNAL1;                 ┐
  ....                           │
  SEND BUFFERED SIGNAL2;         │ a subprogram
  ....                           │
  EXIT;                          ┘

  ENTER SIGNAL3;                 ┐
  ....                           │ a subprogram
  SEND DIRECT SIGNAL4;           ┘

  CUSELESS = 0;

  ENTER SIGNAL5;                 ┐
  ....                           │
  SEND BUFFERED SIGNAL6;         │ a subprogram
  ....                           │
  SEND DIRECT SIGNAL7;           ┘

  ENTER SIGNAL8;                 ┐
  ....                           │ a subprogram
  EXIT;                          ┘

  ....
  END PROGRAM;
```

Figure 3.3: *A PLEX program file divided in subprograms. Note that the assignment* `CUSELESS = 0;` *will never be executed since it is placed between an exit and an enter statement. (See also Fig. 2.6 where a complete program file is described.)*

Since the exchange handles several calls simultaneously while the CP can only execute one program at a time, the CP must queue the signals somewhere. This is done in job buffers, a job table or in time queues and this will be explored in Section 3.2.

As was said earlier there are different parameters that describe the signal properties of a CP-CP signal. Three groups classify these properties and each signal has one property from each group. Each group is described below and all possible combinations is shown in Fig. 3.7.

3.1.1 Direct and buffered signals

As was stated in Section 3.1, the main distinction between (CP-CP) signals is whether they are direct or buffered. Buffered signals start **a new** job, whereas direct signals **continue** the current job. (Jobs are covered in Section 3.2.1). That is, they are handled differently in the execution model.

Direct signals reach the receiving block immediately, they could be seen as direct jumps to another unit. By using direct signals, other signals have no possibility of coming-in-between, i.e., the programmer *retains control* over the execution. However, direct signals are normally only allowed to be used in very time-critical program sequences, such as call set-up routines.

With buffered signals, it is not predictable when the signal reaches the receiving block. Direct and buffered signals are illustrated in Fig. 3.4.

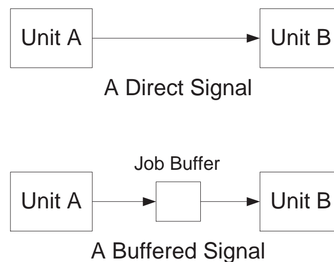


Figure 3.4: *Direct and buffered signals.*

3.1.2 Unique and multiple signals

This distinction concerns the number of receivers of the signal. A unique signal can only be received in one particular block, while a multiple signal can go to any block as shown in Fig. 3.5. However, it is not possible to send a multiple signal to more than one block simultaneously which means that a multiple signal **does not** perform multicast[4]. But even if a multiple signal can go to any of the receiving blocks specified in

[4]Multicast: Send once - received by all

the *Signal Survey*[5], the signal sending statement must always contain one (**and only one**) receiver of the multiple signal.

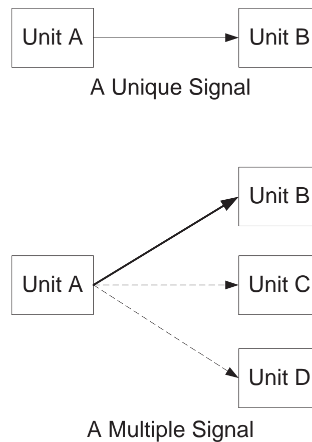


Figure 3.5: *Unique and multiple signals.*

3.1.3 Single and combined signals

The third distinction concerns whether the sending block expects an answer. Combined signals demand an immediate answer, while single signals do not require such feedback. For this reason, combined signals can **never** be buffered (as shown in Fig. 3.7). Instead, they behave like direct jumps from one unit to another. When the execution in the other unit (the receiver of the signal) finishes, execution jumps back to the originating unit. Combined signals are always direct signals, which means that execution continues without interrupt and all other signals have to wait. Fig. 3.6 illustrates these kind of signals.

When discussing the sending and receiving of combined signals, one will also mention *forward* and *backward* signals. A communication between two parts[6] is always initiated by one of the parts. The initiating part is sending the forward signal whereas the part that replies to the call is sending the backward signal. This is pictured in Fig. 3.8.

[5]The Signal Survey is described in Section 2.2.1

[6]Which, in our target domain, is the sending and receiving of signals between *function blocks*.

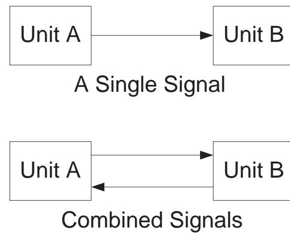


Figure 3.6: *Single and combined signals.*

Signal Type		Direct	Buffered
Single	unique	X	X
	multiple	X	X
Combined	unique	X	
	multiple	X	

Figure 3.7: *Possible properties for CP-CP signals. X indicates a legal/possible combination, shaded with Grey indicates an illegal alternative.* **NOTE**: *A combined* **backward** *signal can not be multiple since this signal is an answer (i.e., an acknowledgment) to a "caller" and must therefore return to the "caller" and nobody else.*

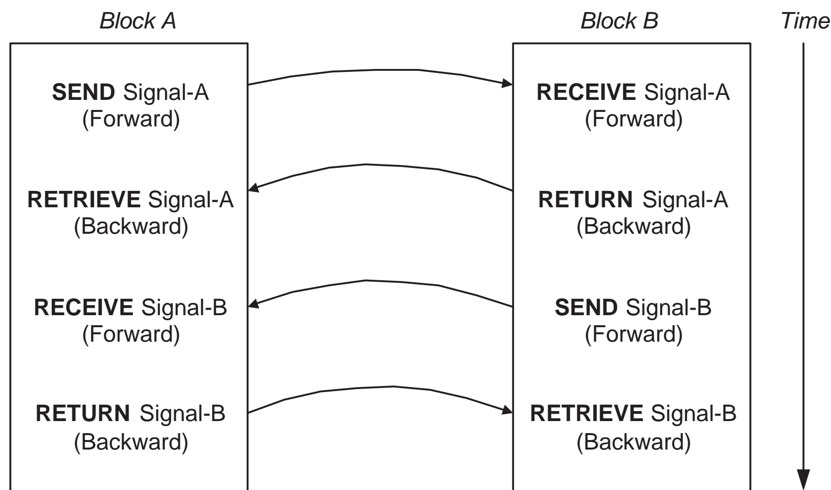


Figure 3.8: *Forward and Backward signals.*

3.1.4 Local and Non-local signals

In the beginning of Section 3.1, we said that signals are used ”for the interwork between software units”. But signals can also be used for the interwork between *different parts of the same software unit*. These signals are called *local signals*, since they are local to the software unit they belong to. I.e., the recipient resides in the same software unit. (Consequently, all other signals are called *non-local* signals.

The behavior of a local signal is similar to that of a GOTO statement since they result in direct jumps to the recipient. (And in that sense, they can be regarded as direct signals.)

Whether a signal is local or not, is specified in the Signal Description (which was briefly explained in Section 2.2.1, and covered in more detail in Appendix B). The distinction between local and non-local signals is of importance in, for instance a semantic framework for PLEX.

3.1.5 Signals and Priorities

Every signal that is sent in the system is assigned a priority level, A - D. The priority level is of importance when the signal is to be buffered (Section 3.2), and it tells the ”importance” of the source code that is triggered to execution by the signal. The priority of each signal is specified in the corresponding Signal Description.

3.1.6 Signals and Data

Signal Data are variable values sent with a signal[7]. The data may consist of field variables, symbol variables, pointers, numerals, string objects, buffer variables and field expressions. For single and combined signals, it is possible to send 25 signal data. The data is loaded to the register memory in the central processor (see Section 2.1.1) if the signal is direct, **or** to the job buffer if the signal is to be buffered.

3.2 Jobs, Signal Buffers and Job Handling

In the following sub-sections, we will discuss the definition of a *job* (Section 3.2.1), the different ways of delaying/buffering a signal (Section

[7]This is similar to a *call by value* function call.

3.2.2) and, finally, how jobs are handled at runtime (Section 3.2.3).

3.2.1 What is a Job?

A job is a continuous sequence of statements executed in the processor. A job begins with an `ENTER` statement for a **buffered** signal and ends with an `EXIT` statement.

Between the `ENTER` and the `EXIT` statement, several buffered signals (or no signals at all) may be sent. A job is not limited to one CP software unit, several units and blocks can take part in a job.

A job does **always** have a single entry point but it *may* have multiple exit points.

In Section 3.1.5 we discussed the priority of a signal. In the following subsections, we will instead talk about the priority of a job. This make sense since it is more natural to look at whole jobs when discussing execution of PLEX code, than it is to look at a single[8] signal. The reason is that a job includes the actual PLEX code that is triggered to execution by the signal, as well as the signal itself.

3.2.2 Signal Buffers

Some jobs in the AXE system are not time-critical and can wait to be executed, while others need to be executed immediately. The first case holds for administrative jobs and the second case for jobs related to traffic handling (i.e., telephone calls[9]) and CP faults.

Buffered signals (which could be read as ”the start of a new job”) may be delayed using one of the following methods:

- Job Buffer: delays a signal until all ”older” jobs have been processed
- Job Table: sends signals at short periodic intervals
- Time Queue: delays signals by relative or absolute time

We will look further to these different ways of delaying a signal.

[8]By single signals, we do **not** mean single signals as described in Section 3.1.3.

[9]A normal load on the system is 200 telephone calls that is to be handled every second. These jobs are all time critical and have the same priority, but the performance would not be acceptable with a ”*first-come-first-served*” approach. A solutions is to use buffered signals as a ”time sharing” mechanism.

Job Buffers: Job buffers are queues with a **FIFO-semantics**[10]. There are four buffers for CP-CP and RP-CP signals and one for CP-RP signals; *Job Buffer A*, *Job Buffer B*, *Job Buffer C* and *Job Buffer D*, all for CP-CP and RP-CP signals, where Job Buffer A has the highest priority. *Job Buffer R* is the buffer for CP-RP signals.

The buffers carry the following type of tasks:
Job Buffer A - *urgent tasks of the operating system*; preferential jobs, e.g., errors in traffic equipment.
Job Buffer B - telephone traffic.
Job Buffer C - I/O communication. The command statement described in Section 2.1.3 is handled at this level.
Job Buffer D - APZ routine self-tests.
Job Buffer R - CP-RP signals queue in JBR, a buffer for signals sent from the CP to a RP.

The Job Table: The job table contains jobs executed at short periodic intervals, for instance, incrementing clocks for time supervision. The job table has higher priority than any of the job buffers. Since the possible execution time after a job table signal is very short, this signal only initiates a program sequence in the receiving block, which inserts a buffered signal in one of the job buffers. The buffered signal initiates the "real" work in the program which from an application point of view, has the priority of the buffer it is inserted in.

Time Queues: Time queues delay periodic and other jobs at longer intervals than the job table. There is one absolute time queue and three relative ones. The absolute time queue stores the absolute time for signal execution (month, day, hour and minute). Every minute, the time queue compares this value with the system calendar. When there is a match, the signal is moved to one of the four job buffers. The three relative queues have a counter for each job. Every 100 ms, 1 second and 1 minute, respectively, the time queue receives a periodic signal from the job table and decrements the counter. If a counter reaches the value zero, the corresponding signal is forwarded to one of the job buffers. I.e., a signal that

[10] First In First Out

is fetched from a time queue is almost never executed at once[11]. Execution of the signal is performed when the operating system fetches it from the job buffer it was inserted in.

Fig. 3.9 shows how a software unit sends a delayed (and multiple) signal. The signal is first placed in a time queue and after that in a job buffer. After it is taken from the job buffer, the execution is started in the receiving unit.

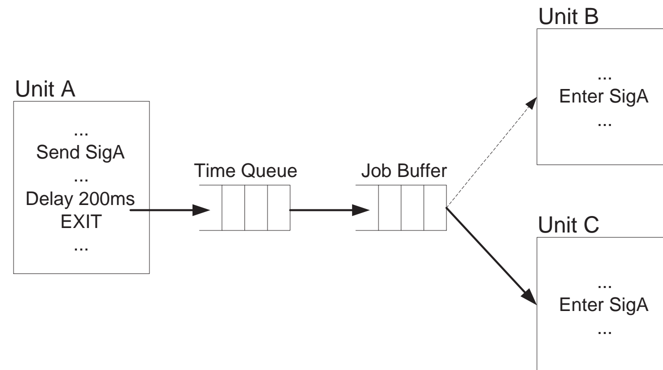


Figure 3.9: *Sending of a delayed (and multiple) signal. The signal is sent from Unit A and received in Unit C but, as could be seen in the figure, it is possible to receive the signal in Unit B as well **if** Unit B is specified as the receiver by the PLEX designer.*

3.2.3 Job Handling

The priorities at runtime correspond to the priorities among the job buffers (Section 3.2.2), as will be shown below.

As already stated, Section 3.2.2, depending on their purpose and time requirements, jobs are assigned to certain priority levels - five different levels exist. But the important thing, when discussing job priorities, is how different priority levels can interrupt each other and, as could be seen in the following discussion, we could view the five different priority levels as only three if we take the possibility for one job to preempt another into consideration.

[11]The only exception is when the receiving job buffer (and every job buffer with higher priority) is empty.

Tasks initiated by a periodic Job Table signal use the traffic-handling level 1 (THL 1), JBA signals use traffic-handling level 2 (THL 2), JBB use traffic-handling level 3 (THL 3), JBC use base level 1 (BAL 1) and JBD use base level 2 (BAL 2), see Fig. 3.10.

The Job Table has a higher priority than all the job buffers. JBA has a higher priority than JBB, and so forth. The jobs in the job buffers are executed in order of priority - JBA is emptied before JBB, and so on. Data used in interrupted jobs stay in the processor register memory, and THL, BAL 1 and BAL 2 jobs have their own processor registers. That means **all** THL jobs share the same register buffers. Hence, no job at one sub level of THL can interrupt a job at another sub level of THL, since they share the same set of registers and the temporary variables would be destroyed otherwise.

I.e., jobs from the job table, JBA and JBB have to wait for each other, but all three can interrupt job from JBC and JBD. As BAL 1 and BAL 2 have different register memories, JBC can interrupt JBD.

JBA - urgent tasks of the operating system: preferential traffic
JBB - all other telephone traffic
JBC - input/output to operator and I/O devices
JBD - APZ routine self-test
JBR - signals from Central Processor to Regional Processor
THL - traffic-handling level
BAL - base level

Figure 3.10: *Job buffers and runtime priorities in the AXE system.*

In some cases, however, it may be necessary to prevent the system from interrupting an important task. For example, an operation and maintenance (O&M, Section 2.1.3) routine at C-level (BAL 1) is writing to variables that are also accessed by traffic-handling routines at B-level (THL 3). In this situation, it is best to inhibit the interrupt function as

long as the writing at C-level is in progress. The interrupt function is inhibited by the `DISABLE INTERRUPT` statement and activated by the `ENABLE INTERRUPT` statement.

We conclude this subsection with an exampel. Fig. 3.11 illustrates the execution of several jobs. In the figure, the execution starts in block 1 with the first job, proceeds in block 2 with the second job and finally ends in block 1 with the execution of the last job. Fig. 3.12 gives a closer look of the link (into job buffers) and execute process.

If a new job enters an empty job buffer, the buffer sends an interrupt signal for that priority level. If the ongoing job has a lower priority level, that job is interrupted. However, a job can not interrupt a job on the same (or higher) priority level.

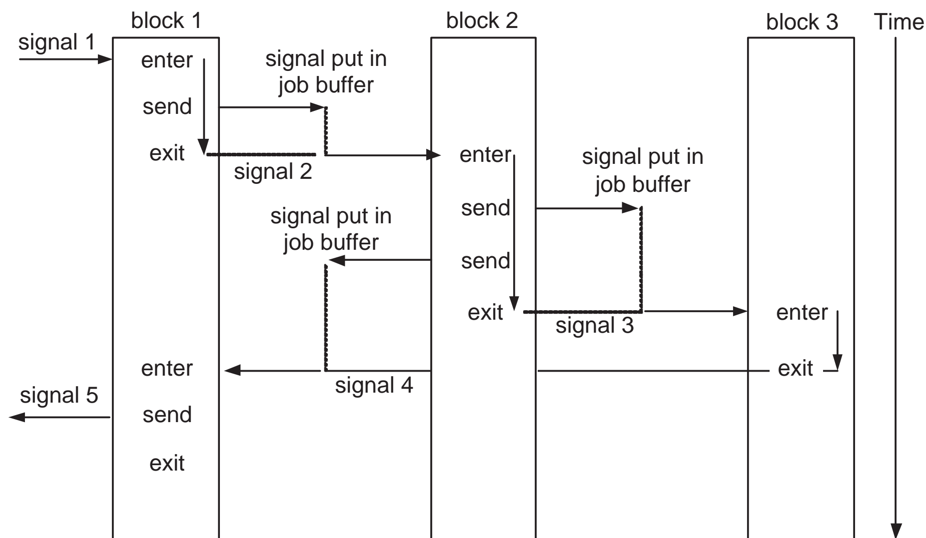


Figure 3.11: *The execution model - Four jobs are executed. The process of transfering a buffered signal from the sending block to the receiving, via a job buffer, is shown in Fig. 3.12. NOTE the "parallel" architecture that could become real parallel execution.*

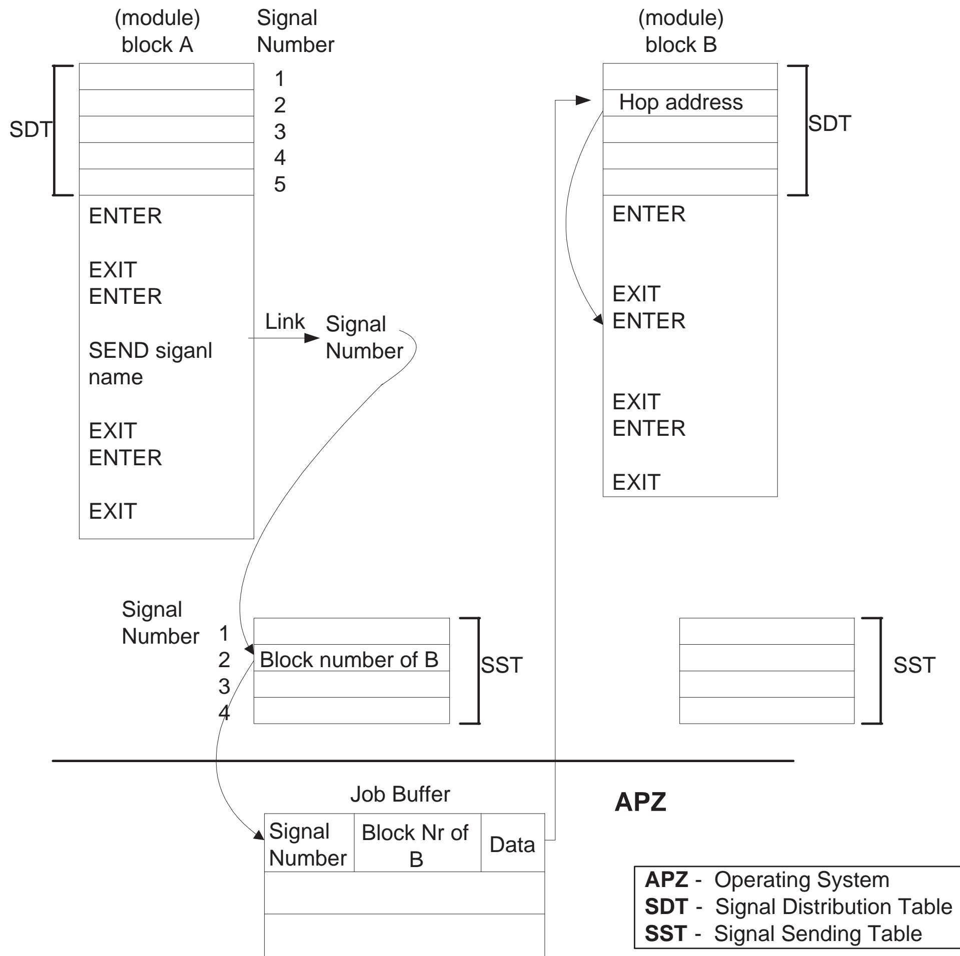


Figure 3.12: *Linking and execution for a buffered signal in APZ. See also Fig. 3.11.* **NOTE**: *The procedure is the same for direct signals except that they **not** are inserted in a Job Buffer.*

3.2.4 Execution Time Limits

As stated in Section 3.0.5, PLEX is a real-time language. This means that a system programmed in PLEX is a real-time system[12]. When talking about execution times limits, one always refer to the execution time of a job. There are limits for the execution time, but this is not measured in absolute times. Instead, there are programmer guidelines that specify how many lines of code that may be placed in a software unit (or units) for one job.

3.3 Linking Encapsulation

All blocks used in the system are compiled separately and it is also possible to "load" them separately, even at run-time. This process is called a Function Change and it was described in Section 2.1.1. When doing a Function Change, the Signal-Sending Table (SST) and the Global-Signal Distribution Table (GSDT) has to be updated. The update has to be done because all signal sendings has to look in the SST and the GSDT to find which signal to invoke.

When updating the tables, by the Rationalized Software Production (RSP) functionality, the (new) introduced signal is given a unique number, the Global Signal Number (GSN). This number is stored in the GSDT as well as in the SST of the Function Unit (block) using this "new" signal. The GSDT also stores Block Number Receiving (BN-R), (the unique number of the block receiving the signal) and the Local Signal Number (LSN) which is the position holding the local relative address of the entry point of the signal entry.

The Signal Distribution Table (SDT) is *not* updated, as the SDT holds the relative address to the signal entries inside the Function Unit. SDT is set with a local number in the object step (during compilation).

SDT: Contains the relative entry address, set during compilation, of the specific program sequences where signals are received.

SST: Contains the global signal number (GSN) of signals to invoke

[12] And, as shown by Arnström et. al, the AXE system is classified as a *soft real-time* system [AGG99].

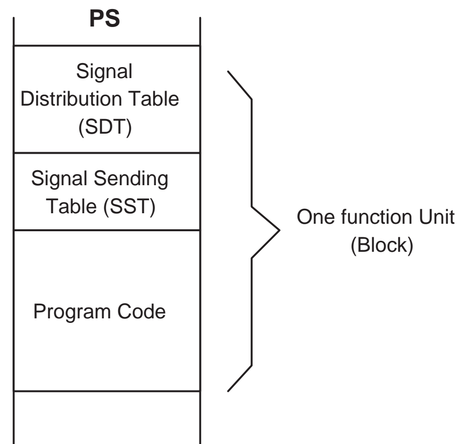


Figure 3.13: *PS, showing SDT, SST and Program Code of one function unit.*

from function unit using "this" SST, created in the object step and changed by the RSP.

GSDT: Contains the global signal number (GSN), the Block Number Receiving (BN-R) and the Local Signal Number (LSN).

In DS, values are stored for all variables.

In PS, the programs for all blocks are stored together with the Signal-Sending Tables (SST), the Signal Distribution Table (SDT) and the Global-Signal Distribution Table (GSDT), see Fig. 3.13

RS is used for addressing DS and PS, and contain the Program Start Address (PSA) and Base Start Address (BSA), see Fig. 3.14.

3.3.1 Addressing a Program Sequence

Fig. 3.15 shows "unit A" sending a signal to "unit B"; the global signal number (GSN) is found in the Signal-Sending Table (SST) of "unit A". The GSN is used to find the Block Number Receive (BN-R) and the Local Signal Number (LSN) in "unit B" ("unit A" doesn't know it is "unit B" that holds the signal entry for the signal sent from "unit A"). The BN-R is used to obtain the Program Start Address (PSA) in the Register Store (RS). The PSA is an absolute address in the Program Store (PS), and by knowing the LSN and PSA, and also using the Signal Distribution Table

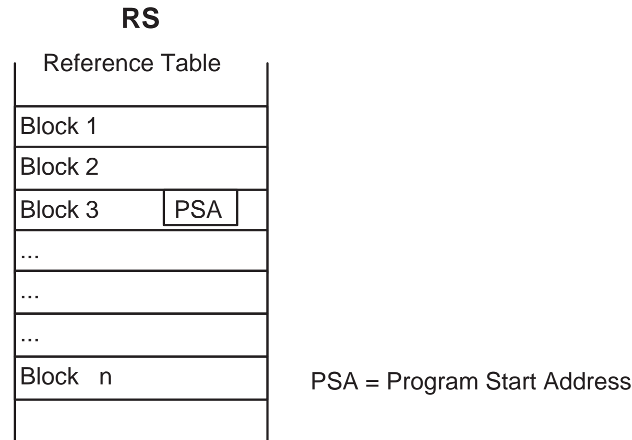


Figure 3.14: *RS, showing the Reference Table.*

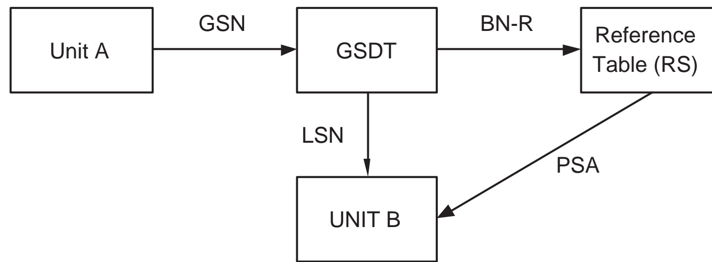


Figure 3.15: *The information flow in determining the signal entry when sending a signal.*

(SDT) of ”unit B” the entry point of the program code can be determined in ”unit B”. See Fig. 3.16.

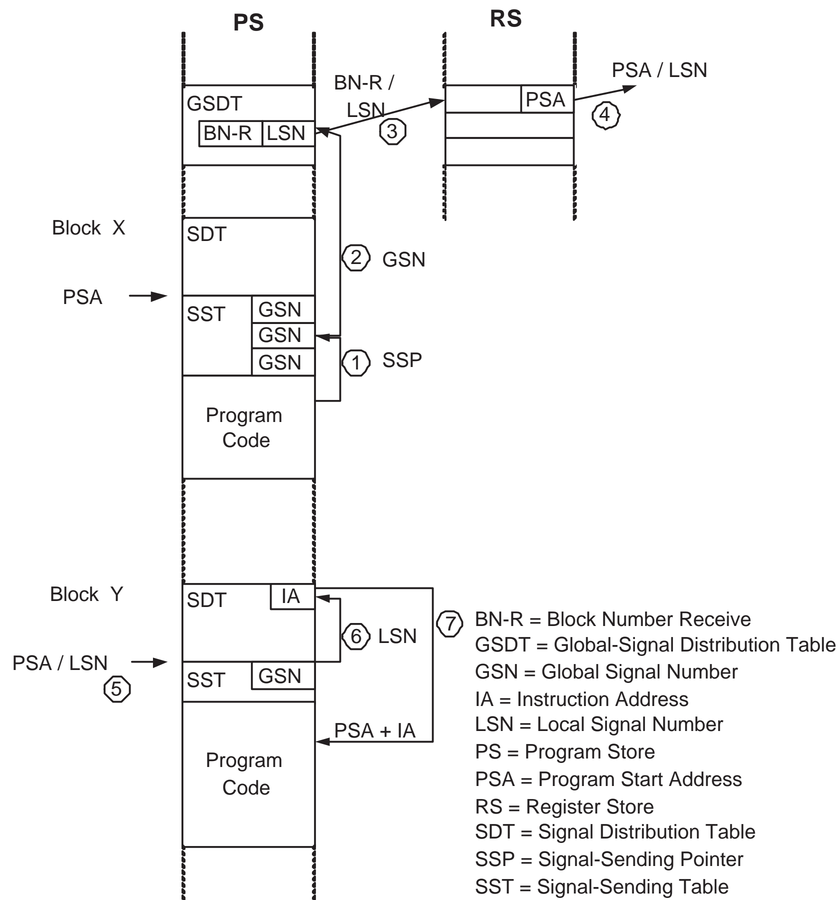


Figure 3.16: *The consecutive order of handling a signal sending.*

Addressing in DS

RS actually consists of two parts: the Reference Table (RT) and the Base Address Table (BAT). In the RT there is one PSA and Base Start Address (BSA) for each block, and the BSA points to the starting point of BAT, see Fig. 3.17[13].

RT: is part of Register Store and hold the Program Start Address and Base Start Address.

BAT: holds the address of the variables in DS. For each block a variable is given a number from 1 and upwards. This number is called the *Base Address Number* (BAN). To get the address of a variable in DS, the BSA + BAN will give the position holding the address in DS.

BSA: holds the address of current start point of BAT.

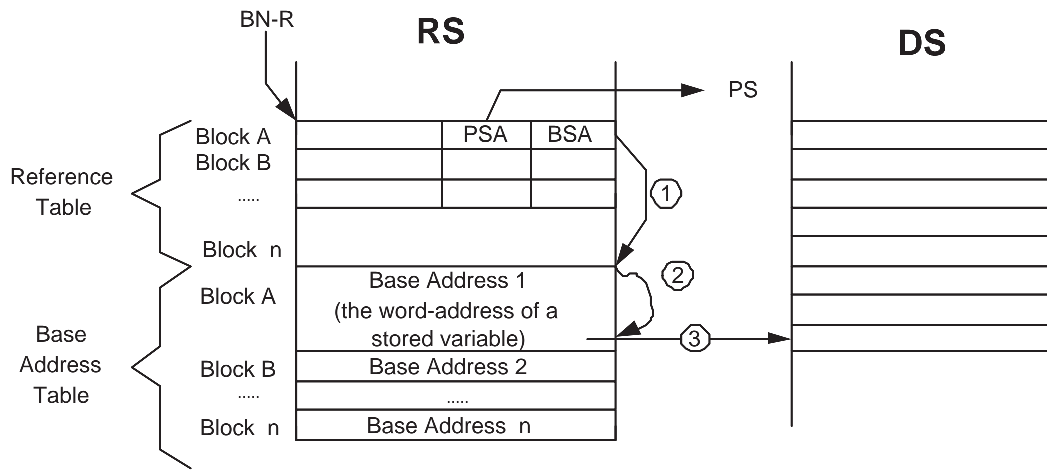
3.4 Software Recovery

After the initial loading (Section 2.1.4), the exchange is supposed to run smoothly during its lifetime. This is also the normal situation for the system. However, errors can't be entirely eliminated and in this section we will study software recovery actions. The goal with the automated recovery action is to minimize the exchange down-time. This is achieved by first trying to release only the dysfunctional forlopp[14] (which normally stretches over parts of several blocks) and leave the rest of the system unaffected. As a last step, if nothing else works, the entire system is restarted.

This section will cover the different steps regarding software recovery actions. In Section 2.2.3 we stated that variables are treated differently at recovery actions depending on the properties set by the designer. We will end this section with a summary of variable properties and their "behavior" at recovery actions.

[13]Actually, this is how addressing is performed in some architectures. The addressing principles may differ among the APZ versions

[14]Forlopp will be described in Section 3.4.1



1 = BSA indicates the starting point of base address table for block A located in reference store. The BSA will give the absolute address.
2 = BAN indicates where the BAT word address for the specific variable is found. BAN is a relative address.
3 = The word address indicates where the value of the specific variable is stored in DS.

Figure 3.17: *Show how addressing to DS is performed in RS. BSA points to the starting point of BAT*

3.4.1 Forlopp

The first line of defense for maintaining system availability is the *Forlopp* release. The purpose of a forlopp release is to allow a single process chain, e.g., a call, to be released without adversely affecting any other processes in the system.

Forlopp originates from the Swedish word ”förlopp” meaning ”sequence of related events”. In the contents of AXE, a typical forlopp will result in a ”path through the system” which generally will be represented by a chain of linked software resources, such as records. In AXE, the word forlopp can be used to denote both the ”sequence of related events” and the resulting ”path through the system”. The forlopp mechanism is implemented in the Maintenance Subsystem, MAS, Fig. 2.1. Examples of forlopps are an ordinary telephone call or a command. Some concepts associated with forlopps:

- A forlopp identity (FID), stored in a special register, is assigned to each process (a call or forlopp). All parts participating in the same forlopp have the same forlopp identity.
- The forlopp manager (FM) stores information concerning the different forlopps.
- When a software error is detected, the FM sends *release signals* to the blocks involved according to the information stored in FM. A forlopp release is hereby performed.
- At a forlopp release, a software error dump is performed, which means that the contents of the records participating in the current forlopp are dumped[15].

To summarize, a detected software fault may result in a forlopp release, provided that the function block in which the fault occurred is *forlopp-adapted* and the forlopp function is active.

3.4.2 System Restart

The *system restart* has been the traditional recovery action taken by the APZ (Section 2.1) when it detects a software fault. The system restart

[15] Section 2.1.4 describes what a dump is.

affects the entire system and not only the forlopp in which the fault occurred. The purpose of a system restart is to restore the system to a predefined state.

During restart, *restart signals* are sent to each block, so that during successive restart phases, blocks perform actions to complete the initialization or restoration to a consistent value of their data store variables.

The system restart procedure could be initiated manually, by a `COMMAND` (Section 2.1.3), or automatically. A manual system restart clears error situations, for instance the disconnection of a hanging device. An automatic system restart is detected by programs, microprograms and supervisory circuits. At a system restart, the job table, the job buffers and the time queues (Section 3.2) are cleared.

There are three levels of system restart activities:

- Small system restart, which does not affect calls in speech position and semi-permanent connections. Other calls are disconnected. This is a minimal system restart.
- Large system restart in which all calls are disconnected. Semi-permanent connections are not affected.
- Reload and large system restart in which a reload is performed first to ensure that `RELOAD`-marked variables contain correct values. This is then followed by a large system restart. Semi-permanent connections are disconnected and automatically reestablished.

The reason to have different types of system restarts is to disturb traffic handling as little as possible during the restart phase.

With the occurrence of the first fault in a normal block that leads to a system restart, the system tries to repair itself without disturbing the traffic too much - A small system restart is initiated. If another serious fault occurs within a predefined time interval, a large system restart will be initiated. In the event of the occurrence of a third serious fault within another predefined time interval, a reload and a large system restart will take place. This represents the system's most extreme error-recovery action. The described phases is pictured in Fig. 3.18.

Finally, it is sometimes unnecessary to immediately initiate an automatic system restart. The system restart could be delayed or inhibited. This is done by calling the *selective restart* function.

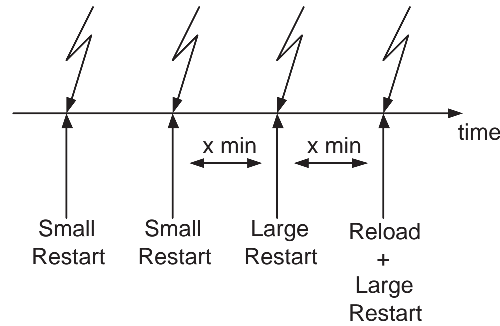


Figure 3.18: *Different types of system restart.*

3.4.3 Forlopp Release or a System Restart?

In Section 3.4.1 we described the concept of forlopps as a way to recover from a software error without affecting more than the faulty forlopp. Then, in the following Section, 3.4.2, we described the system restart and the different levels of restart and when they apply. This section explains when the system restart action takes over from the forlopp release mechanism.

As we said in Section 3.4.1, a forlopp release is always a first choice if an error has been detected. The system restart "function" applies when and if:

- The forlopp release fails to recover the system (i.e., the faulty forlopp), or
- The faulty process has not been forlopp-adapted, or
- The number of faults have been to high according to a predetermined limit.

The last case is checked against an *intensity counter*. This counter keeps tracks of the quantity of software faults. The counter is stepped each time a fault is detected leading to a delayed system restart or a forlopp release. When the counter reaches the predetermined limit, a system restart is initiated. The counter is then reset and starts again from zero. Fig. 3.19 shows the intensity counter and Fig. 3.20 shows the different levels of software recovery.

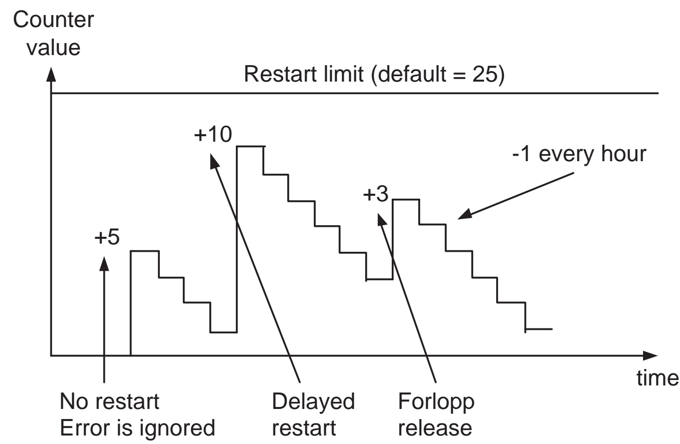


Figure 3.19: *The intensity counter.*

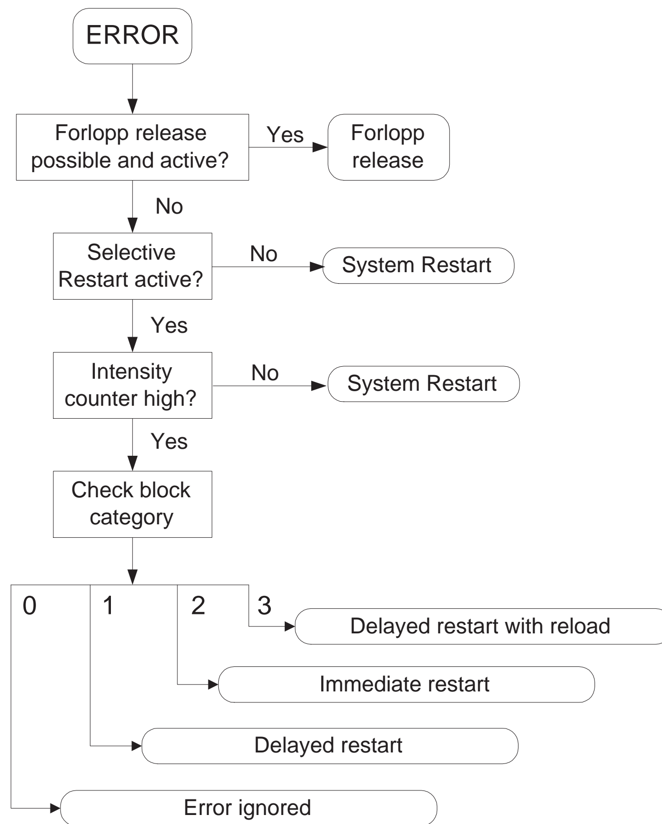


Figure 3.20: *Different levels of recovery after a detected software error.*

3.4.4 Variables and Software Recovery

As said in Section 2.2.3, the variable properties determine how the variables are to be treated (i.e., from a data point of view) in the case of a system restart. Fig. 3.21 shows the principles of how different types of variables should be treated after a system restart.

<table>
<tr><th></th><th>Start</th><th>Small system restart</th><th>Large system restart</th><th>System restart with reloading</th></tr>
<tr><td>DS
DS DUMP
DS STATIC</td><td>Cannot be trusted</td><td colspan="2">Cannot be trusted
Exception: when the variable value is checked in system restart routine</td><td>Cannot be trusted</td></tr>
<tr><td>DS RELOAD
DS RELOAD DUMP
DS RELOAD STATIC</td><td colspan="3">Can be trusted</td><td>Can be trusted</td></tr>
<tr><td>DS CLEAR
DS CLEAR DUMP</td><td colspan="4">Can be trusted</td></tr>
</table>

Figure 3.21: *Data security of different start/restart types.*

Part II

Semantics

Chapter 4

Related Work

In Section 1.1, we mentioned that efforts have been made to replace PLEX with, or "map" PLEX to, specification languages like SDL for instance. The latest version of SDL (Specification and Description Language), *SDL2000*, has been given a formal (semantic) definition[1]. Work on this can be found in [IT00] and [EGG^{+}01]. Whether or not SDL and PLEX are "compatible" with each other is still discussed within the PLEX design group, and will not be a subject of study in this report.

Another language used for specification is UML, Unified Modeling Language. The relation between PLEX and UML has been discussed in [AGG99]. A dialect of UML, UML-RT, is discussed by Herzberg in [Her99]. This paper investigates the mapping between PLEX and UML-RT. This paper is published within the *E-CARES* project[2], which also explores the AXE system, but from a "re-engineering" point of view.

Another language used in the same domain, i.e., the telecommunication domain, is CHILL (CCITT High Level Language), which was developed by the International Telecommunication Union (ITU), [IT99]. The language has been specified by the Vienna Development Method, [BJ82], in [IT82]. This is a denotational framework and could be of interest since CHILL is used in the same domain as PLEX. But, since the VDM is more of a specification method, that goes from abstract notation

[1] See the home page of the *SDL Formal Semantics Project* on:
http://rn.informatik.uni-kl.de/projects/sdl/

[2] Home page:
http://www-i3.informatik.rwth-aachen.de/research/projects/ecares/Main.html

to formal specification, whereas this work goes from an implemented language to a formal specification, the VDM will not be considered in this report.

CHILL has also been covered by Winkler in [Win00].

Chapter 5

Programming Language Semantics

What is programming language semantics and how/when is it used? The first section in this chapter is devoted to answer these questions. We will explain what we mean by a semantic description of a programming language. The following sections will look at the most common forms of semantics.

5.1 The meaning of a program

Programming language semantics is concerned with rigorously specifying the meaning, or *the effect*, of programs that are to be executed. By effect we mean, for instance, the contents of the memory locations, which parts of the program that is to be executed, or the behavior of the hardware affected by the program. A semantic specification captures these things in a formal way, and later in this chapter we will study different approaches to this formal description.

Formal descriptions of programming languages are becoming more and more popular, e.g. the BNF[1] is used to specify the syntax of PLEX. The problem is that a formal description of the syntax says nothing about the meaning of the program since ”syntax is concerned with the grammatical structure of programs” whereas ”semantics is concerned

[1] Backus-Naur Form

with the meaning of *grammatically correct* programs" [NN92]. It is also possible to distinguish between *static* and *dynamic* semantics [Mos01]. Static semantics corresponds to the compile-time behavior of the program, the dynamic semantics hence corresponds to the run-time behavior.

But now when we know what programming language semantics is, how can we use it? We consult [NN92] for an answer to this question:

- The semantics can reveal ambiguities and complexities in what may look as a clear documentation of the language (e.g. the language manual).
- The semantics can also form the basis for implementation, analysis and verification.

Even with these obvious advantages with formal semantics, it is still widely regarded as being of interest only to theoreticians [Mos01].

5.2 Semantic approaches

The formalizations of a programming language may differ, i.e. the explanations (or the meaning) can be formalized in different ways. Most frameworks, or approaches, can be classified as one of the following three categories:

- *Operational semantics - How to execute the program.* The operational approach(es) is (are) concerned with *how* the effect of the computations is produced. The meaning is often specified by an abstract machine and/or a transition system.
- *Denotational semantics - The effect of executing the program.* Denotational semantics, in contrast to operational semantics, is only concerned with the effect of the computation, not how it is obtained. Meanings are modeled by mathematical objects representing the effect of executing the constructs.
- *Axiomatic semantics - Partial correctness properties of the program.* The axiomatic approach is concentrated on specific properties of a

program. These properties are expressed as *assertions*. Axiomatic semantics involves rules for checking these assertions. There may be aspects of the executions that are ignored since only specific properties are considered.

These different categories will be further investigated/explained in their own sections (5.4, 5.5 and 5.6), but we can already state that in the general case, the semantics will tell us something about the relation between an initial and a final *state*[2].

5.3 Notation

In the following studies of the different approaches, some "semantic" notations will be used[3]. These notations may seem unfamiliar to the reader and we will summarize some basic notations here.

- By $[x \mapsto 5]$ we mean the ***function*** that *maps* the *symbol* x to the numerical value 5. I.e. x *has* the value 5.
- $s_0 = [x \mapsto 5]$ is called a *state*. The state s_0 where x maps to 5. We will also use the general form $s\ x$ to denote the variable x in state s. For a value 3, bound to x in the state s, we will write $s\ x$ = **3**.
- By $\langle x := z, s_0 \rangle \rightarrow s_1$ we mean that the execution of x:=z in the state s_0 will result in the new state s_1.
- We will later talk about *semantic functions* and use the notation $[\![\ ]\!]$ to denote the *syntactic argument* (enclosed in syntactic brackets) to the semantic function.
- We will use *tt* and *ff* to denote the *truth values* **true** and **false**, respectively.
- $\frac{\langle\ S_1,\ s\ \rangle\ \rightarrow\ s',\ \langle\ S_2,\ s'\ \rangle\ \rightarrow\ s''}{\langle\ S_1;\ S_2,\ s\rangle\ \rightarrow\ s''}$ is called a *rule*. It has a number of *premises* above the solid line and one *conclusion* below the solid line. A rule may also have a number of *conditions* that have to be fulfilled whenever the rule is applied.

---

[2]The meaning of a state is explained in the following subsection.
[3]The notation used in this report are the same as used in [NN92]

- We will use $\Rightarrow$ as the *transition relation*, i.e., when we go from one state to another with the execution of **one** statement.

- $\Rightarrow^*$ means that we take *zero or more* steps from an initial to a final state.

5.4 Operational Semantics

In Section 5.2 we said that a semantic formalism tells us something about the relationship between the initial and the final state. In the *Operational approach* we are not only interested in this relationship. We also want to know how the computations that lead to the final state modify the *intermediate state(s)* as well. The different operational approaches differ in the level of details:

- In the *Natural semantics*, the focus is on how the overall results of the executions are obtained, whereas

- in the *Structural operational semantics*, it is of interest in how the individual steps of the execution takes place[4].

These two different approaches will be studied in the following subsections (5.4.1 and 5.4.2). For both styles of operational semantics, the *meaning* of statements is specified by a transition system with two different configurations:

- $\langle S, s\rangle$ which means that the *statement S* is to be executed from the *state s*.

- *s* that represent a terminal state, i.e. a termination of the computation.

5.4.1 Natural Semantics

The relationship between the initial and the final state of the execution of a statement is in focus for the *natural semantics* (ns). The transition system specifies this relationship for every statement and is written in

[4]Natural semantics is also known as *Big Steps Semantics*, whereas the structural operational approach sometimes is called *Small Steps Semantics*. These alternative names tell us something about how detailed the transition is specified.

the form $\langle S, s\rangle \rightarrow s'$ which, intuitively, means that the execution of the statement S from state s will *terminate* and the resulting state will be s'. The execution of the statement S on the state s will:

- terminate iff[5] there is a state s' such that $\langle S, s\rangle \rightarrow s'$, and provided that the language under consideration is deterministic, or
- loop iff there is no such state s'.

An example on how a natural semantics may look is given in Fig. 5.1[6]. In this table, we see a compound statement consisting of the individual statements S_1 and S_2. We see that the execution of S_1 from s results in a final state s', a final state that also is a start state in the configuration $\langle\ S_2,\ s'\ \rangle$ which has the final state s''. So, the execution of the compound statement $S_1;\ S_2$ from the start state s result in a final state s''.

$$[comp_{ns}] \quad \frac{\langle\ S_1,\ s\ \rangle \rightarrow s',\ \langle\ S_2,\ s'\ \rangle \rightarrow s''}{\langle\ S_1;\ S_2,\ s\rangle \rightarrow s''}$$

Figure 5.1: *The compound statement expressed in natural semantics.*

5.4.2 Structural Operational Semantics

In the natural semantics, we were interested in the relationship between the initial and the final state when a statement was executed. In the *structural operational semantics* (sos), we are also interested in the *intermediate* states as well. I.e., if the execution of the statement S in state s leads to the final state s', the structural operational semantics will tell us something about the intermediate states that are "visited" during the execution. In other words, the focus is on the individual steps of the execution. The transition system has the form $\langle S, s\rangle \Rightarrow \gamma$ where γ

---

[5] if **and only** if

[6] This, and other examples of semantic styles/approaches in Section 5.2 are all from [NN92].

is *either* of the form $\langle S', s' \rangle$ **or** of the form s'. This means that the transition system expresses the **first** step of the execution of the statement S from the state s and the result of this is γ.

- If γ is of the form $\langle S', s' \rangle$ then the execution of S from s is **not** completed and the rest of the computation is expressed by the intermediate configuration $\langle S', s' \rangle$.
- If γ, on the other hand, is of the form s', then the execution of S has terminated and the final state is s'.

In Fig. 5.2 we express the compound statement in a structural operational semantics style. Unlike the natural semantics in Fig. 5.1, we see two "cases" for the compound statement. In the first case, the execution is not completed and the next configuration (from $\langle S_1, s \rangle$) is the intermediate configuration $\langle S_1', s' \rangle$. In the second case, the execution of S_1 is completed and the final state is s'. From this state, the execution of the configuration $\langle S_2, s' \rangle$ can start.

$$[comp\ {}^{1}_{sos}] \quad \frac{\langle\ S_1,\ s \rangle \Rightarrow \langle\ S_1',\ s'\ \rangle}{\langle\ S_1;\ S_2,\ s\ \rangle \Rightarrow \langle\ S_1';\ S_2,\ s'\ \rangle}$$

$$[comp\ {}^{2}_{sos}] \quad \frac{\langle\ S_1,\ s \rangle \Rightarrow s'}{\langle\ S_1;\ S_2,\ s\ \rangle \Rightarrow \langle\ S_2,\ s'\ \rangle}$$

Figure 5.2: *The compound statement expressed in structural operational semantics.*

5.5 Denotational Semantics

With the operational approach, the focus was on how the program was executed - "*How to compute.*" In the denotational approach, the emphasis is on "*the effect of computing*". By effect, we mean the relationship between the initial and the final state. The denotational approach consists of defining semantic functions for each syntactic category. This

semantic function maps each syntactic construct to a *mathematical object*, most often a function, that describes the effect of executing the construct. In Fig. 5.3, we see a denotational description of the compound statement (previously shown in Figure 5.1 and 5.2). In the Figure, we see that the effect of computing the compound statement $S_1;\ S_2$ is the *functional composition* of executing S_1 followed by S_2.

$$S_{ds}[\![S_1;\ S_2]\!] \ = \ S_{ds}[\![S_2]\!] \ \circ \ S_{ds}[\![S_1]\!]$$

Figure 5.3: *The compound statement expressed in denotational semantics.*

5.6 Axiomatic Semantics

With the operational as well as the denotational approach, we are concerned with the meaning of a given program. This is in contrast to the *axiomatic* approach were we study *properties* of a given program. These properties (expressed as *assertions*) could, for instance, be execution times (see [NN92] for an example) or the degree of resource utilization. But, since the focus is on specific properties and not on the meaning (as in the previously described styles) "*there may be aspects of the execution that are ignored*" [NN92]. Axiomatic definitions is often given in the form

$$\{P\}S\{Q\}$$

where P is a Pre-condition, S the statement to be executed and Q a Post-condition. This is to be interpreted as: "*If P holds **and** the execution of S terminates, **then** Q will hold*". In Fig. 5.4 we show the compound statement in the axiomatic style. From this, we can say that if the pre-condition P holds and the execution of S_1 followed by the execution of S_2 terminates holds, then the post-condition R will hold.

$$S_p[\![S_1; S_2]\!] \qquad \frac{\{P\}S_1\{Q\},\ \{Q\}S_2\{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

Figure 5.4: *The compound statement expressed in a axiomatic semantics style.*

Chapter 6

Semantic Approach

6.1 Selected Approach and Motivation

In the process to determine a proper semantic approach for PLEX, it was early agreed that the axiomatic approach (described in Section 5.6) was not a suitable choice because of the rather abstract notation and also because its focus is on properties more than the effect. So, the selection had to be made between some form of operational semantics, or the denotational approach. Some considerations that were taken into account:

- The operational semantics has its main advantages[1] in the fact that it is relatively easy to understand and also that the executional steps are explicit. A possible disadvantage may be the fact that the approach is tightly connected to *how* a statement is executed (i.e., it is closer to the actual execution than the denotational approach).

- For the denotational approach, the advantage is that since mathematical objects are in focus, it abstracts away from how programs are executed. It can therefore be seen as the denotational approach provides a more formal basis for reasoning about programs. The disadvantage lays in the notation - Mathematical notation may ”scare” the unfamiliar reader.

[1]This is the authors opinion.

[NN92] claims that the denotational approach may be preferred when *reasoning about* programs whereas the operational approach may be preferred when *implementing* the language. The decision to use an *operational approach* was based on the following considerations:

- It was a pre-request that the notation would be relatively easy to understand even by a reader unfamiliar with semantic notation.
- The language PLEX is continuously updated and new hardware is introduced in periodic intervals. This means that our semantic description also could be seen as a *reference manual* for new (future) implementations of the language.

A second decision had to be made whether we should use the *natural* semantics (described in Section 5.4.1) or the *structural operational* semantics (described in Section 5.4.2). Even if the PLEX language does not provide parallelism today, it is shown that many constructs are suitable for parallel execution [Lin03], and to be able to deal with this in a future extension of the (language and the) semantics, the structural operational approach were preferred[2].

6.2 The State of the System

In order to develop a semantic framework for PLEX, we have to determine what a *state of the system* is since the execution of a statement will change the current state, i.e., with the execution of a statement, we go from an old state to a new state. Our starting point will be:

- With *the state of the system*, we mean *the contents of the memory*.

So, in order to determine the state of the system, we will look at the different forms of storage that exist in the system.

The first thing we have to consider is the memory and stores of the Central Processor. The memory in the Central Processor (CP) consists of the register memory and the different stores.

[2] [NN92] shows why the structural operational semantics is well suited for parallel constructs.

The register memory is used for storing of temporary variables, pointers and for transferring of signal data between different "processes"[3]. Values in the registers are lost after an EXIT from a *job*, or other register killing statements such as direct signals, combined signals or IO statements, for example. These are all statements that force the execution to leave the current *block*. Temporary variables are only "alive" (i.e., containing a value in a register) from the first write to it, until the last use. The PLEX compiler keeps track of write-read chains and only allocates registers while the variables are alive during the execution.

The different stores in the CP were shown in Fig. 2.2. They are repeated in Fig. 6.1. The different forms of storage contains the following information:

- The *Program Store* is used for the storage of programs.
- Variables that is to survive the termination of a "process" or a system restart, are stored in the *Data Store*.
- The *Reference Store*, finally, contains information about where to find the different programs and data.

However, the Program Store and the Reference Store will be omitted when we specify the state of the system since the statements we look at (starting in Section 6.12) don't effect these storages. The second thing

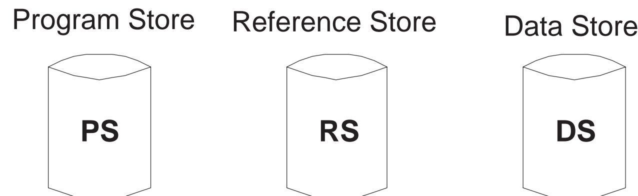


Figure 6.1: *The different stores in the central processor (CP).*

that is important, is the different kind of job buffers (which are used to store the different kinds of signals). A simplified figure of the job buffers is shown in Fig. 6.2[4]. The semantics of the job buffers are the FIFO[5]

[3]The transferring of signal data have similarities to an ordinary function call: Just as variables are loaded when the function is entered, are the signal data loaded at the start of a *sub-program*, i.e., at a SIGNAL ENTRY.

[4]A more detailed description is found in Fig. 3.10.

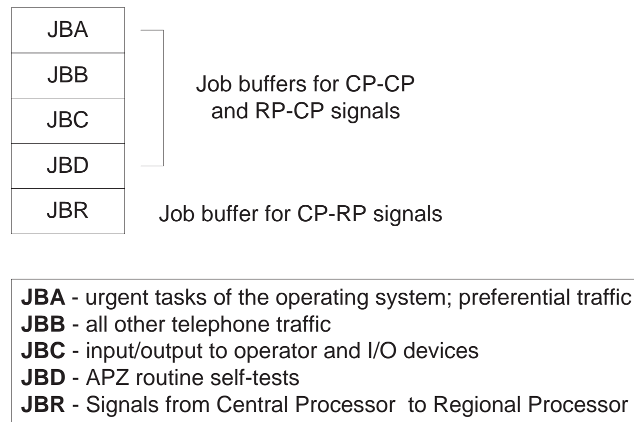


Figure 6.2: *A simplified figure of the job buffers in the PLEX/AXE environment. (See also Fig. 3.10)*

approach, as shown in Fig. 6.3. Associated with each job buffer are *two pointers*, ***Job Buffer In*** *(JBI)* and ***Job Buffer Out*** *(JBO)*. These pointers make it possible for the APZ (the operating system) to know where, in a job buffer, to insert a signal that is to be buffered, or from where to fetch a new signal. The memory layout, in a job buffer, for **one** buffered signal is pictured in Fig. 6.4. The job buffer will consequently consist of several items of the form described in the picture. So, as

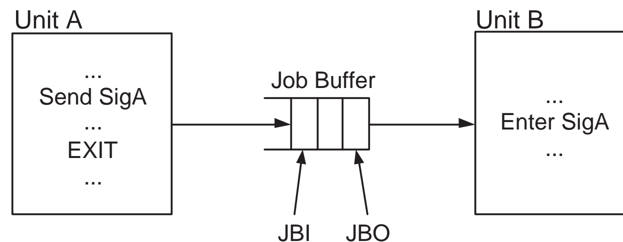


Figure 6.3: *The FIFO-semantics of the job buffers.*

stated in the beginning of this section, our starting point in determining the state of the system, was:

- With *the state of the system*, we mean *the contents of the memory*.

[5]First-In-First-Out

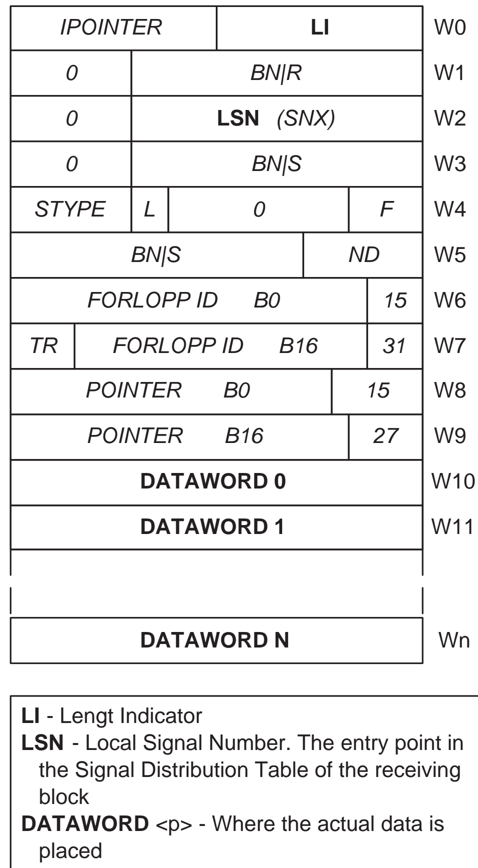


Figure 6.4: *Organization of the job buffers. NOTE: The pointer registers, W8-W9, are treated as one register (and referred to as POINTER B0). This is due to the fact that the register in W9 is only used when a pointer is too large to fit only in W8. In this case, the pointer will be split up and placed in W8 with its first half, and in W9 with its second half.*

And, we can now tell that the contents of the memory is the contents of the register memory, the contents of the Data Store, and the contents of the job buffers. The state of the system can, at this point, be represented as a tuple of the form[6]:

$$\langle RM, DS, JBA, JBB, JBC, JBD, JBR \rangle$$

where RM = Register Memory, DS = Data Store, JBA = Job Buffer A, JBB = Job Buffer B, JBC = Job Buffer C, JBD = Job Buffer D and JBR = Job Buffer R.

The above representation would indeed be sufficient if PLEX was a strictly sequential language, i.e., if $statement_A$ precedes $statement_B$ in our source code file, then we could be absolutely certain that $statement_A$ would execute before $statement_B$. This holds in a language without the possibility to *manipulate the Program Counter*, e.g., with a `JUMP` statement. But since these kind of statements **do** exist in PLEX[7], we also have to model the possibility of ”moving” the execution to somewhere different from the following statement. If we had been dealing with some kind of assembler code, the above could easily be managed by adjusting the Program Counter according to the *instructions* executed by the CPU. But we are not looking at the semantics from an instruction level point of view, instead we are working with statements that consists of n instructions.

- Our solution is to define the ***Virtual Statement Counter***, $\mathcal{VSC}$

The $\mathcal{VSC}$ contains an address (an integer value), which tells where the first *instruction* in the next *statement* resides in memory. By stepping up the $\mathcal{VSC}$ with 1, we step it up to the first instruction in the next statement[8]. If nothing else is stated, the $\mathcal{VSC}$ will **always** be stepped up to the next statement after the execution of a previous statement, i.e., $s[\mathcal{VSC} \mapsto \mathcal{VSC}{+}1]$. For convenience, we will use the notation $s[\mathcal{VSC}{+}{+}]$

[6]The contents of the memory could also be extended with the contents of the `DATA SECTOR` (see Section 2.2.1), which is used in a startup phase as well as in a system restart phase. But since we do not cover those phases in this report, the contents of the Data Sector is omitted in our description of the systems state.

[7]E.g., the `GOTO` statement.

[8]The approach is similar to the one described in [Lis98], where the *program counter* is modified in a similar way.

to denote $s[\mathcal{VSC} \mapsto \mathcal{VSC}{+}1]$. In situations where this is obvious, we will omit this "information". A situation where this is the case would for instance look like: $\langle S_1, s\rangle \Rightarrow \langle S', s'\rangle$

The above leads us to the following definition of the state of the system:

$$\langle \mathcal{VSC}, RM, DS, JBA, JBB, JBC, JBD, JBR\rangle$$

6.3 Lexical Units and Syntactical Categories

In this section[9], we will follow the conventions in [AB02] and use the same variant of EBNF (Extended Backus-Naur Form) notation.

- ::= is used as the "definition operator".
- We will mark a **KEYWORD** with bold, upper case letters.
- The vertical bar, |, represents a choice between elements.
- The square brackets [ ] are used to denote an optional construct. I.e., the construct inside the brackets may be omitted.
- Optional properties are marked in a separate way, like in "`length`".
- Curly brackets, { }, are used to delimit an operand, or simply to improve readability. $\{\ \}^*$ represents a possibly empty sequence of elements, i.e., the element may be repeated zero or more times. Finally, $\{\ \}^+$, are used for the *non-empty* sequence of elements.

The lexical units and the syntactic categories[10] in PLEX are as follows:

- *upper-case-letter* ::= `A|B|C|...|X|Y|Z`
- *lower-case-letter* ::= `a|b|c|...|x|y|z`
- *letter* ::= *upper-case-letter* | *lower-case-letter*
- *decimal-digit* ::= `1|2|3|...|7|8|9`
- *hexadecimal-digit* ::= `1|2|...|8|9|A|B|...|E|F`
- *special-character* ::= `!|"|#|%|1|'|?|(|)|*|+|,|-|.|/|:|;|<|=|>|_||[|]|$|@`

[9]The **syntactical** material in this section is basically compiled together from the following internal Ericsson documents: [AB98, AB02, AB86]

[10]Please note that the syntactical categories in this section are only a *subset* of the language.

- *character* ::= *letter* | *decimal-digit* | *special-character*
- *identifier* ::= *letter* {*letter* | *decimal-digit* | _{ *letter* | *decimal-digit*}}*
- *decimal-numeral* ::= *decimal-digit*{*decimal-digit* | _}*
- *hexadecimal-numeral* ::= *hexadecimal-digit*{*hexadecimal-digit* | _}*
- *numeral* ::= [+]{*decimal-numeral* | *hexadecimal-numeral*}
- *string-object* ::= "*character*{$character$}*"
- *arithmetic-operator* ::= '+' | '-' | '*' | '/'
- *bit-operator* ::= '(-)' | '(*)' | '(+)' | '(=)' | '=>' | '<='
- *relation-operator* ::= ' < ' | ' =< ' | ' = ' | ' /=' | ' >= ' | ' > '
- *binary-operator*[11] ::= *arithmetic-operator* | *bit-operator*[12] |
  relation-operator

  † There are no syntactic categories such as arithmetic or boolean expressions in PLEX. Instead, there is the *field-expression*. A field expression is one or several *operands* separated by arithmetic or "logical"[13] operators[14]. We could say, "separated by binary operators", but that would include bit operators that don't "give" logical values. Field expression is evaluated from left to right and according to the priorities shown in Table 6.1.

- *field-expression* ::= [+ | -] *sub-expression*
- *subexpression* ::= *operand* | *negation-operator sub-expression*
  | *subexpression binary-operator subexpression*
  | '('*subexpression*')'
- *operand* ::= *field-variable* | *pointer* | *numeral* | *number-symbol*

  † Operands and (sub)expressions can **only** adopt positive integers [AB02]!

- *field-variable* ::= *identifier*

  † When a field-variable is declared, the *variable properties* must also be specified. The variable properties is concerned with if the variable is stored or temporary, and how the variable is to be treated

[11]The meaning of, and the priority among the operators are summarized in Table. 6.1.

[12]All bit operators are binary operators ***except*** the NOT-operator '(-)'.

[13]With logical operators, we mean operators that are used in expressions that we think of as being *true* or *false*, and these operators are marked as logical in Table 6.1.

[14]I.e., arithmetic and boolean expressions exist, but are both called *field expressions*.

during software recovery actions. The property is specified as a *valid*[15] combination of the keywords: **DS**, **BUFFER**, **COMMUNICATION BUFFER**, **CLEAR**, **SEMICLEAR**, **RELOAD**, **DUMP**, **STATIC** and **TRANSIENT**

- *pointer* ::= **POINTER** *pointer-name* '(' *record-name* ')' ';'
  - *pointer-name* ::= *identifier*
  - *record-name* ::= *identifier*

  † [AB98] states that the *pointer-name* should end with the suffix `pointer`, `ptr` or `p`.

  †† Pointers are not more than record numbers (i.e., positive integers). A pointer is always associated with a file and the value of the pointer is the number of the current record in that file. (See Section 2.2.2 for details.)

  ††† Pointers behave like, and are treated as, temporary variables, i.e., pointers are stored in the register memory and they may loose their value whenever the execution of the software unit terminates.There is no explicit pointer arithmetic, instead they are treated and used as operands in *field-expressions* (described above) and in assignment statements (described later in this section).

- *number-symbol* ::= *identifier*

  † Number symbols have the same semantic meaning as *constants* in `C` (for example).

  †† Number symbols are declared in the *Declare Sector* (see Section 2.2.1) with the statement:
  NSYMB *identifier* '=' {*decimal-numeral* | *hexadecimal-numeral*} ';'

- *condition* ::= *field-expression relation-operator field-expression*
  | *symbol-variable* ['=' | '/='] *symbol-value*
  | *string-object* ['=' | '/='] *string-object*

- *symbol-variable* ::= **SYMBOL VARIABLE**
  '(' *symbol-value* {, *symbol-value*}* ')' [`properties`] ';'
  - *symbol-value* ::= *identifier*
  - `properties` = A valid combination of: **DS**, **BUFFER**, **COMMUNI-**

[15] The valid combinations of the variable properties, as well as the meaning of the different properties is covered in Section 2.2.3.

CATION BUFFER, **CLEAR**, **SEMICLEAR**, **RELOAD**, **DUMP**, **STATIC** and **TRANSIENT**

† Symbol variables are also known as *enumeration type* variables, a concept that is familiar to `C` programmers. Their use is to hold *symbol* values, which in the AXE domain could be `IDLE`, `BUSY`, `BLOCKED` for instance. A symbol variables can be assigned symbol values and compared to other symbol variables.

†† The code generation phase[16] replaces the symbol values with numerical values. This is why symbol variables can be assigned to, and compared with, field variables (that store numerical values), i.e., if a field variable is assigned a symbol value, the numerical value of the symbol is assigned the field variable.

Priority	**Operator**	**Meaning**	**Type**
1	`(-)`	Logical NOT	Bit operator (logical)
2	`=>`	Shift right	Bit operator
2	`<=`	Shift left	Bit operator
3	`*`	Multiplication	Arithmetic
3	`/`	Division	Arithmetic
4	`+, -`	Addition, Subtraction	Arithmetic
5	`=`	Equality	Relation (logical)
5	`/=`	Inequality	Relation (logical)
5	`<`	Less than	Relation (logical)
5	`>`	Greater than	Relation (logical)
5	`=<`	Less or equal than	Relation (logical)
5	`>=`	Equal or greater than	Relation (logical)
6	`(*)`	Logical AND	Bit operator (logical)
7	`(=)`	Logical XOR (Exclusive or)	Bit operator (logical)
8	`(+)`	Logical OR	Bit operator (logical)

Table 6.1: *Priorities and meaning of the PLEX operators. Priorities are numbered from the highest (1) to the lowest (8).*

[16]The code generation phase is the process of translating (in several steps) the source code (PLEX) to machine code. (The compilation process, in which the code generation takes place, is covered in [AE00].)

6.4 Statements for Variable Assignments

The general form of the assignment statement has the following syntax:

- *assignment-statement* ::= [**SET**] *variable* '=' *expression* ';'

We recall (from Section 2.2.3) that PLEX have three different types of variables; *Field variables* for numerical information, *symbol variables* for symbolic values, e.g., `IDLE, BLOCKED, BUSY`, etc and *string variables* which store text strings. This implies the following three statements for variable assignment:

- *field-assignment-statement* ::= [**SET**]
  { { {*pointer* | *field-variable*} '=' }$^+$
  {*field-expression* | *maxnum-expression*} }
  | { *field-variable* '='}$^+$ *symbol-variable* ';'

  - *maxnum-expression* ::= **MAXNUM OF** *variable*

  † The value of a maxnum-expression is the number of individuals in the file associated with the given variable name.

- *symbol-assignment-statement* ::=
  [**SET**] {*symbol-variable* '='}$^+$ *symbol-value* ';'

- *string-assignment-statement* ::=
  [**SET**] {*string-variable* '='}$^+$ *string-object* ';'

6.5 Jump Statements

PLEX offers both conditional and unconditional jump statements. The "programmer guidelines" for the use of jump statements are: "*Please avoid backward jumps as far as possible, because they are difficult to follow.*" and "*Try to avoid jump statements, since they can give rise to a poor program structure. It is very difficult to read a program with a great many jump statements.*" [AB98]. But still, they are part of the language, and we will look at their syntax here.

The unconditional jump statement simply interrupts the sequential execution order and its syntax is as follows:

- *unconditional-jump-statement* ::= {**GOTO** | **GO TO**} *label* ';'

The conditional jump statement offers a possibility to jump if the given condition is met. It has the form of an `if`-statement with a `goto`:

- *if-statement* ::= **IF** [**NOT**] *condition* [**PROCEED ELSE**]
  {**GOTO** | **GO TO**} *label* ';'

Since *condition* will return in other statements, its syntax is given here:

- *condition* ::= {*field-expression*$_1$ *relation-operator field-expression*$_2$}
  | {*symbol-variable* {'=' | '/='} *symbol-value*}
  | {*string-object* {'=' | '/='} *string-object* }

In connection to the jump statements, there is also the `BRANCH` statement which selects between a number of program sequences, but since this statement has been "removed"[17] from the language, we will not cover it here.

6.6 Conditional Statements

The `if`-statement in Section 6.5 is a rather primitive construction since it does not have an `else` clause and because the only permitted action is an unconditional jump. This is the motivation for the following more recent statement:

- *improved-if-statement* ::=
  IF [**NOT**] *condition* **THEN** *sequence-of-statements*
  {**ELSEIF** [**NOT**] *condition* **THEN** *sequence-of-statements*}*
  [**ELSE** *sequence-of-statements*]
  FI

[17] I.e., it is an *old statement* and should be avoided in new design [AB98].

6.7 Selections

The `CASE` statement is used for selection between an arbitrary[18] number of unambiguous choices:

- *case-statement* ::=

  CASE *expression* **IS**
  {**WHEN** *choice* {',' *choice*}* **DO** *sequence-of-statements*}+
  OTHERWISE DO *sequence-of-statements*
  ESAC ';'

- *expression* ::= *field-expression* | *symbol-variable* | *string-object*

- *choice* ::= *numeral* | *number-symbol* | *symbol-value* | *string-symbol* | *text-string*

It is not allowed to use a signal reception statement (Section 6.9) inside the sequence of statements part[19]. Neither is it allowed to jump into a `CASE` statement.

6.8 Iterations

The well known `While` statement is missing in PLEX. The main reason is that such a construct is not considered necessary in the application domain (which is the central part of the AXE switching system). A second reason is that the construct *may* give rise to unpredictable execution times, which is **not** the case with the iteration statements in PLEX (see below).

Instead, PLEX offers three different statements for iteration: `ON`, `FOR ALL` and `FOR FIRST`. They are all used for scanning files or indexed variables ***between given start and stop values***. The three iteration statements are compared in Fig. 6.5 and their syntax are as follows:

- *on-statement* ::=

  ON {*pointer* | *field-variable*} **FROM** *field-expression*
  {**UPTO** | **DOWNTO**} *field-expression*

[18] I.e., **one** or more

[19] Which also holds for iterations and conditionals.

DO *sequence-of-statements*
NO ';'

- *for-statement* : : =
  FOR {**FIRST** | **ALL**} {*pointer* | *field-variable*}
  FROM *field-expression-1* [**UNTIL** *field-expression-2*]
  [**WHERE** { [**NOT**] *condition*
  | *field-variable* **IS CHANGED TO** *field-expression*}]
  { {**GOTO** | **GO TO**} *label*
  | **DO** {*statement* | *statement-block-name*} } ';'

Criterion	**ON**	**FOR ALL**	**FOR FIRST**
Ascending or descending order	Yes	Always descending	
Several statements in action part	Yes	No, except in statement blocks, IF, CASE and loop statements	
Condition in iteration statements	NO	Possible	Always
Iteration ends after matching condition once and handling one individual	Not applicable	No	Yes
Iteration variable/pointer after loop	Undefined	Undefined	Defined if matching individual/condition
Generates high-speed loop	No[20]	Possible	Possible

Figure 6.5: *PLEX iteration statements - a comparison.*

6.9 Signal Sending/Receiving Statements

In Section 3.1 it was stated that the main distinction to be made between the different kinds of signals in PLEX is between *direct* and *buffered* signals. And this is true *from the point of view of the execution model*, but from a *syntactical* point of view, the main distinction is between *single and combined* signals. Every signal will fall into one

of these categories. I.e., from the syntactical point of view it is only of interest whether one has to wait for an "answer" (a combined signal) or not - Whether the signal is direct or buffered, or unique or multiple,[21] is specified in the *Signal Description*, described in appendix B, and is certainly of interest from a ***semantic*** point of view.

6.9.1 Statements for Single Signals

There are two statements that are concerned with single signals - The first one given is the statement for *sending* a single signal, whereas the second *receives* a single signal:

- *single-signal-transmission-statement* ::=

```
SEND signal [REFERENCE field-variable]
   [WITH signal-datum {',' signal-datum}*]
   [ [','] { BUFFER
         | HURRY
         | DELAY {numeral
                    | field-variable
                    | field-expression}
                    {MS | S | M}
         | DELAY UNTIL {numeral | field-variable}} ]';'
```

NOTE: The reference "variable" must be given if the signal is a *multiple* signal, which is stated in the Signal Description. The "**WITH** *signal-datum*" part is the data that the signal is carrying. Current programming guidelines says that the keyword **BUFFER**, which states that the signal is to be buffered, should not be used in new design [AB98]. The remaining keywords is concerned with if the signal is to be buffered or not (**HURRY**), and if it is to be buffered, for how long (**DELAY**...). However, if the signal is to be buffered or not, **also** depends on the information given in the Signal Description, and this will be covered in more detail in Appendix B.

- *single-signal-reception-statement* ::=

```
ENTER signal [WITH signal-datum { ','
   signal-datum}* ]';'
```

[20]Except for `BUFFER CLEAR/COPY` (Section 2.2.3).

[21]The different signal types is described in Section 3.1

NOTE: When a signal is fetched from a job buffer, this statement **normally** marks the entry point in the corresponding code that is to be executed[22].

6.9.2 Statements for Combined Signals

- *combined-forward-signal-transmission-statement* : : =
  SEND *cfsignal* [**REFERENCE** *field-variable*]
  [**WITH** *signal-datum* {',' *signal-datum*}*]
  [',']
  WAIT FOR $cbsignal_1$ **IN** *label*
  {**OR** $cbsignal_n$ **IN** *label*}* ';'

NOTE: As could be seen in the syntax for the sending of a combined forward signal, it must be specified where (in the code) the "answer" is supposed to arrive. Also, note that it is allowed to wait for more than one backward signal. This is useful when one wants to take action according to the result of the execution in the block that received the forward signal (and then sent the backward signal).
The following statement is used for receiving of a forward signal.

- *combined-forward-signal-reception-statement* : : =
  RECEIVE *cfsignal*
  [**WITH** *signal-datum* {',' *signal-datum* }*] ';'

The following two statements is concerned with the answer to the initiating part. The first statement is sending the "reply" whereas the second receives the "answer".

- *combined-backward-signal-transmission-statement* : : =
  RETURN *cbsignal*
  [**WITH** *signal-datum* {',' *signal-datum*}*] ';'

- *combined-backward-signal-reception-statement* : : =
  label')' **RETRIEVE** *cbsignal*
  [**WITH** *signal-datum* {',' *signal-datum*}*] ';'

[22] The entry point for code execution could also be the statement for receiving a combined (forward) signal (described in Section 6.9.2).

6.9.3 Statements for Local Signals

The following statements is for the sending, and receiving of a local signal, respectively. (Local signals are described in Section 3.1.4.)

- *local-signal-sending-statement* ::=
  TRANSFER *signal*
  [**WITH** *signal-datum* {',' *signal-datum*}*] ';'

- *local-signal-receiving-statement* ::=
  ENTRANCE *signal*
  [**WITH** *signal-datum* {',' *signal-datum*}*] ';'

6.10 Exit

The exit statement is a *deactivation* statement for the sequence of statements that begins with a signal reception statement for a single signal or a command reception statement.

- *exit-statement* ::= **EXIT** ';'

NOTE: The exit statement marks the (possible) termination of a subprogram and also the termination of a job, since the control is transferred back to the operating system (which fetches the next job from the job buffers).

6.11 Semantic Functions

In order to develop a semantic description of the most important parts of PLEX, which is considered to be the concept of signals (as mentioned in Section 1.2), we need some *semantic functions* [23] [24] that will help us expressing the meaning of the language. Some of them will be defined in this section whereas the other will be defined when the need arise.

[23] Semantic functions are briefly introduced in Section 5.3.

[24] When we, in this section as well as in the following report, denote the state of the system with s, we mean the tuple $\langle \mathcal{V}SC, RM, DS, JBA, JBB, JBC, JBD, JBR \rangle$ that was specified in Section 6.2.

However, we summarize all semantic functions, that we define and use in Chapter 6, in Table 6.2.

Function	**Defined in**
$\mathcal{N} : numeral \rightarrow \mathbf{Z}$	Section 6.11
$\mathcal{ST} : string \rightarrow$ **Characters**	Section 6.11
$\mathcal{A} : field\text{-}expression \rightarrow \mathbf{Z}$	Table 6.3, Section 6.11
$\mathcal{B} : field\text{-}expression \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$	Table 6.4, Section 6.11
$\mathcal{P} : variable \rightarrow var\text{-}property$	Section 6.11
$\mathcal{ADR} : label \rightarrow \mathbf{Z}$	Section 6.11
$\mathcal{STMT} : address \rightarrow$ *statement*	Section 6.11
$\mathcal{SD} : signal \rightarrow sig\text{-}property$	Section 6.17
$\mathcal{SIG} : signal \rightarrow address$	Section 6.17
$\mathcal{L} : signal \rightarrow level$	Section 6.17
$\mathcal{BLOCK} : address \rightarrow CodeBlock$	Section 6.17.2
$\mathcal{APZ} : state \hookrightarrow state$	Table 6.5, Section 6.18
$\mathcal{CODE} : address \rightarrow code$	Section 6.19
$\mathcal{S}_{PLEX} : Code \rightarrow state$	Section 6.19

Table 6.2: *A summary of the semantic functions defined and used in Chapter 6 (and in Appendix A).*

The first function that we introduce is the function that determine the number represented by a numeral, and this function is defined in the following way:

$$\mathcal{N} : numeral \rightarrow \mathbf{Z}$$

This function is to be read as "*The semantic function $\mathcal{N}$ that takes a numeral as input and returns the corresponding integer value*", i.e., a number. An example of applying the function is $\mathcal{N}[\![137]\!] = \mathbf{137} \in \mathbf{Z}$. **NOTE**: It is important to understand the difference between the *numeral* 137, which is a syntactic construct (and therefore enclosed in syntactical braces), and the meaning of the numeral, which is the *number* 137. In the following, we will write $\mathcal{N}[\![n]\!]$ to denote the "meaning" of n, i.e., the corresponding *numerical value*.

The second function is similar to the first, but ”operates” on strings:

$$\mathcal{ST} : string \rightarrow \textbf{Characters}$$

and returns the meaning of a *string-object* as in:

$$\mathcal{ST}[\![\text{abc}]\!] = \textbf{abc} \in \textbf{Characters}$$

In other words, both $\mathcal{N}$ and $\mathcal{ST}$ captures what we intuitively think of when we talk about the meaning of *syntactical constructs* such as '137' or 'abc'.

The third function we will use, is the function that will determine the meaning of a *field expression*. We recall, from Section 6.3, that PLEX does **not** contain the ”usual” arithmetic and boolean expressions. Instead, there is the field expression. But depending on the operator, the value of a field expression could be seen as numerical **or** logical. We will therefore define **two** semantic functions for field expressions, one that applies when the expression is to be treated as an arithmetic expression, and one that applies when the expression is to be treated as a logical expression. The first function, $\mathcal{A}$, summarized in table 6.3, is defined in the following way:

- The meaning of a field expression, when the expression is a *numeral* as in $\mathcal{A}[\![\text{n}]\!]s$, is given by the function $\mathcal{N}$, as specified above[25].
- The meaning of a variable x is the value currently bound to the variable, $s\ x$.
- The meaning of a field expression consisting of *sub-expressions*, $\mathcal{A}[\![a_1 + a_2]\!]s$, is determined by the meanings of the sub-expressions.

For field expressions that is to be treated as logical expressions, their semantics is given by the semantic function $\mathcal{B}$ as specified in Table 6.4[26].

The next function defined, is the function $\mathcal{P}$, which will find the specified properties for the given variable. $\mathcal{P}$ is defined in the following way:

$$\mathcal{P} : variable \rightarrow var\text{-}property$$

[25] Remember that the s after the syntactic braces, $[\![\ ]\!]$, represent the *state of the system* which was specified in Section 6.2.

[26] When specifying the semantics for the Selection statement (Section 6.15), we will have to compare two strings and see if they are equal. This motivates the rule $\mathcal{B}[\![t_1 = t_2]\!]s$ in the table.

$$\begin{array}{lcl} \mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![x]\!]s & = & s\ x \\ \mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1 *\ a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s *\ \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1\ /\ a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s\ /\ \mathcal{A}[\![a_2]\!]s \end{array}$$

Table 6.3: *The semantics of "arithmetic" expressions.*

where $var\text{-}property \subset \{\varepsilon,$ **DS**, **BUFFER**, **COMMUNICATION BUFFER**, **CLEAR**, **SEMICLEAR**, **RELOAD**, **DUMP**, **STATIC** and **TRANSIENT**$\}$[27].

It is necessary to determine the variable properties, since depending on which properties that are specified for the given variable, different parts of the state of the system will be updated in an assignment.

Finally, to be able to determine the start address for a given program label and also to find the statement that is located at a given address, we define the following two functions

$$\mathcal{ADR} : label \rightarrow \mathbf{Z}$$

$$\mathcal{STMT} : address \rightarrow \textit{statement}$$

where $\mathcal{ADR}$ takes a program label and returns its corresponding address, and whereas $\mathcal{STMT}$ takes an address and returns the corresponding PLEX statement.

6.12 The Semantics for Assignment Statements

Now, when the required functions have been specified, we are ready to approach the language constructs that is to be treated in this report. We start with the semantics of the *assignment statements*, since these

[27] ε is returned if the variable is temporary.

[28] This rule could also be expressed as: $\mathcal{B}[\![a_1(=)a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{B}[\![a_1]\!]s \neq \mathcal{B}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{B}[\![a_1]\!]s = \mathcal{B}[\![a_2]\!]s \end{cases}$

$$\mathcal{B}[\![a_1 < a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{A}[\![a_1]\!]s < \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{A}[\![a_1]\!]s >= \mathcal{A}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![a_1 =< a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{A}[\![a_1]\!]s =< \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![a_1 = a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![t_1 = t_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{ST}[\![t_1]\!]s = \mathcal{ST}[\![t_2]\!]s \\ \mathbf{ff} & if\ \mathcal{ST}[\![t_1]\!]s \neq \mathcal{ST}[\![t_2]\!]s \end{cases}$$

$$\mathcal{B}[\![a_1 / = a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![a_1 >= a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{A}[\![a_1]\!]s >= \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{A}[\![a_1]\!]s < \mathcal{A}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![a_1 > a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & if\ \mathcal{A}[\![a_1]\!]s =< \mathcal{A}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![(-)a_1]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{ff} \\ \mathbf{ff} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{tt} \end{cases}$$

$$\mathcal{B}[\![a_1(*)a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![a_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![a_2]\!]s = \mathbf{ff} \end{cases}$$

$$\mathcal{B}[\![a_1(=)a_2]\!]^{28}s = \begin{cases} \mathbf{tt} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![a_2]\!]s = \mathbf{ff} \\ \mathbf{tt} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{ff} \text{ and } \mathcal{B}[\![a_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & if\ \mathcal{B}[\![a_1]\!]s = \mathcal{B}[\![a_2]\!]s \end{cases}$$

$$\mathcal{B}[\![a_1(+)a_2]\!]s = \begin{cases} \mathbf{tt} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{tt} \text{ or } \mathcal{B}[\![a_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & if\ \mathcal{B}[\![a_1]\!]s = \mathbf{ff} \text{ and } \mathcal{B}[\![a_2]\!]s = \mathbf{ff} \end{cases}$$

Table 6.4: *The semantics of "boolean" expressions. (See also, Table 6.1 where the different operators, as well as their meaning, are described.)* a_n *represents a field-expression whereas* t_n *represents a string.*

are the statements that most obviously affect the state of the system by their change of the contents in the memory.

First, from Section 2.2.3, we recall that from a storage point of view, the main distinction to be made between variables is whether they are *temporary* or *stored* which means that they reside in the register memory (RM) or the data store (DS), respectively. To determine this, we use the function

$$\mathcal{P} : variable \rightarrow var\text{-}property$$

which was specified last in the previous section. As was mentioned earlier (Section 6.11) we will use s to denote the tuple

$$\langle \mathcal{VSC}, RM, DS, JBA, JBB, JBC, JBD, JBR \rangle$$

and if one item of this tuple is to be updated, we will write **(DS)**x if x in the DS is to be updated. We will also use the notation **RM** $\mapsto$ `UNDEF` in situations where the contents of the register memory should be considered as non-existing[29]. Secondly, we have three different kinds of variables to deal with; field variables, symbol variables and string variables. The semantics for field variable and symbol variable assignments are similar, but we write them as separate rules to make it explicit which kind of assignment we are dealing with. This leads us to the following rules for assignment statements.:

- [ass$_{field}$] $\langle x := a, s \rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ (DS)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!]$ = DS
- [ass$_{field}$] $\langle x := a, s \rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ (RM)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!] = \varepsilon$

- [ass$_{sym}$] $\langle x := a, s \rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ (DS)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!]$ = DS
- [ass$_{sym}$] $\langle x := a, s \rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ (RM)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!] = \varepsilon$

The final assignment statement concerns string variables and for these, we specify the following rules:

- [ass$_{string}$] $\langle x := c, s \rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ (DS)x \mapsto \mathcal{ST}[\![c]\!]s]$ if $\mathcal{P}[\![x]\!]$ = DS
- [ass$_{string}$] $\langle x := c, s \rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ (RM)x \mapsto \mathcal{ST}[\![c]\!]s]$ if $\mathcal{P}[\![x]\!] = \varepsilon$

---

[29] I.e., we use our own defined constant `UNDEF` in a way similar to the use of `NULL` in C.

6.13 The semantics for Jump statements

As mentioned in Section 6.5, PLEX offers two kinds of jump statements; Unconditional and conditional. The unconditional jump statement will **always** perform a jump to the specified label, whereas the conditional is dependent on the evaluation of the given condition. Their respective semantics are as follows:

- [$\mathbf{jump}_{uncond}$] $\langle \texttt{GOTO}\ label, s\rangle \;\Rightarrow\; s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![label]\!]\ ]$

- [$\mathbf{jump}_{cond}$] $\langle \texttt{IF}\ condition\ \texttt{GOTO}\ label, s\rangle \;\Rightarrow\; s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![label]\!]\ ]$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{tt}$

- [$\mathbf{jump}_{cond}$] $\langle \texttt{IF}\ condition\ \texttt{GOTO}\ label, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\texttt{++}]$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{ff}$

6.14 The semantics for Conditional statements

We begin this section by repeating the syntax for the `if`-statement from Section 6.6. This statement is denoted as the *improved-if-statement* since it improves the above conditional jump statement. As could be seen, from the syntax, the `elseif` as well as the `else` clause are optional. This will be explicit in the semantics for the `if`-statement.

- *improved-if-statement* `::=`

  IF [**NOT**] *condition* **THEN** *sequence-of-statements*
  {**ELSEIF** [**NOT**] *condition* **THEN** *sequence-of-statements*}*
  [**ELSE** *sequence-of-statements*]
  FI

- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1, s\rangle \;\Rightarrow\; \langle S_1,\ s\rangle$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{tt}$

- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\texttt{++}]$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{ff}$

- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2, s\rangle \;\Rightarrow\; \langle S_1,\ s\rangle$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{tt}$

- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2, s\rangle \;\Rightarrow\; \langle S_2,\ s\rangle$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{ff}$

The case `IF...ELSEIF...ELSE` can be seen as an instance of the `IF...ELSE` rule and is therefore omitted.

6.15 The semantics for Selection statements

The syntax for the selection statement, `CASE`, was discussed in Section 6.7 and is repeated below:

- *case-statement* ::=
  CASE *expression* **IS**
  $\{$**WHEN** *choice* $\{$*',' choice*$\}^*$ **DO** *sequence-of-statements*$\}^+$
  OTHERWISE DO *sequence-of-statements*
  ESAC ';'

The pair, *expression/choice*, is either numerical or strings. Both cases are covered by $\mathcal{B}[\![expression = choice]\!]s$ as could be seen from Table 6.4.

We give the semantics in term of the "basic case", i.e., when CASE is followed by one WHEN DO and terminated by a OTHERWISE DO. Multiple instances of WHEN DO could be seen as instances of the "basic case".

- [select] $\langle$`CASE` *expression* `IS WHEN` *choice* `DO` S_1
  `OTHERWISE DO` $S_n, s\rangle \Rightarrow \langle S_1, s\rangle$

  $if\ \mathcal{B}[\![expression = choice]\!]s = \mathbf{tt}$

- [select] $\langle$`CASE` *expression* `IS WHEN` *choice* `DO` S_1
  `OTHERWISE DO` $S_n, s\rangle \Rightarrow \langle S_n, s\rangle$

  $if\ \mathcal{B}[\![expression = choice]\!]s = \mathbf{ff}$

6.16 The semantics for Iteration statements

As stated in Section 6.8, PLEX offers three different statements for iterations; Namely `ON`, `FOR ALL` and `FOR FIRST`. We will give their se-

mantics[30] in turn.[31]

- [ON$_{Up}$]

$$\frac{\langle S,\ s\rangle \Rightarrow s',\ \langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1{+}1\ \texttt{UPTO}\ exp_2\ \texttt{DO}\ S,\ s'\rangle \Rightarrow s''}{\langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1\ \texttt{UPTO}\ exp_2\ \texttt{DO}\ S,\ s\rangle \Rightarrow s''}$$

$$if\ \mathcal{A}[\![exp_1]\!]s\ <\ \mathcal{A}[\![exp_2]\!]s$$

- [ON$_{Up}$] $\langle$ON *pointer/variable* FROM '*expression*$_1$' UPTO '*expression*$_2$' DO $S,\ s\rangle \Rightarrow\ s[\mathcal{VSC}{+}{+}]$

$$if\ \mathcal{A}[\![expression_1]\!]s\ >\ \mathcal{A}[\![expression_2]\!]s$$

The semantics for the ON-statement in the case of DOWNTO instead of UPTO is almost the same in every step, as could be seen in the following rules:

- [ON$_{Down}$]

$$\frac{\langle S,\ s\rangle \Rightarrow s',\ \langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1{-}1\ \texttt{DOWNTO}\ exp_2\ \texttt{DO}\ S,\ s'\rangle \Rightarrow s''}{\langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1\ \texttt{DOWNTO}\ exp_2\ \texttt{DO}\ S,\ s\rangle \Rightarrow s''}$$

$$if\ \mathcal{A}[\![exp_1]\!]s\ >\ \mathcal{A}[\![exp_2]\!]s$$

- [ON$_{Down}$] $\langle$ON *pointer/variable* FROM '*expression*$_1$' DOWNTO '*expression*$_2$' DO $S,\ s\rangle \Rightarrow\ s[\mathcal{VSC}{+}{+}]$

$$if\ \mathcal{A}[\![expression_1]\!]s\ <\ \mathcal{A}[\![expression_2]\!]s$$

Then follows the two FOR-statements, and we start with the FOR ALL. As could be seen from the syntax for the FOR ALL statements (Section 6.8), there are a lot of things that are optional. We omit these parts here. The reason is that these parts only makes it possible to give more precise values for the iteration variables. In its basic form (which will be the form we specify the semantics for), the FOR ALL statements iterate from *pointer/field-expression*$_1$ down to zero.

[30]**Note:** PLEX "offers" the possibility to jump **out** of an iteration with a GOTO statement for example. This means that a proper semantics could not rely on a sequential execution order! However, the following rules does not take that into consideration. I.e., the following rules are specified for *well formed* iterations, which consequently are iterations without such jumps.

[31]For convenience we use *point/var* for *pointer/variable* and *exp*$_n$ for *expression*$_n$.

- $[\text{FOR}_{ALL}]$[32]

$$\frac{\langle S,\ s\rangle \ \Rightarrow\ s',\quad \langle \texttt{FOR ALL}\ pointer/field\text{-}var\ \texttt{FROM}\ field\text{-}exp_1\ \texttt{- 1 DO}\ S,\ s'\rangle \Rightarrow s''}{\langle \texttt{FOR ALL}\ pointer/field\text{-}var\ \texttt{FROM}\ field\text{-}exp_1\ \texttt{DO}\ S,\ s\rangle \Rightarrow s''}$$

$$if\ \mathcal{A}[\![field - exp_1]\!]s\ >\ 0$$

- $[\text{FOR}_{ALL}]$ $\langle$FOR ALL ***pointer/field-variable*** FROM ***field-expression***$_1$ DO ***S/statement-block-name***, $s\rangle \ \Rightarrow\ s[\mathcal{VSC}\text{++}]$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s\ <\ 0$

The difference between the two FOR-statements is that the condition part (see Section 6.8) is mandatory in the FOR FIRST and optional in the FOR ALL. This means that when the condition is fulfilled, if it is, then the FOR FIRST will abort the iteration and transfer the control to the following statements. This will be captured in the following rules for FOR FIRST. Also, as with the FOR ALL statement, we omit parts that are optional and focus on the basic and mandatory parts. The 'action' part in the FOR FIRST statement can either be a GOTO or a DO statement. We will specify this by giving separate rules for these cases.

- $[\text{FOR}_{FIRST}]$ $\langle$FOR FIRST ***pointer/field-variable*** FROM ***field-expression***$_1$ WHERE ***condition*** | ***field-variable*** IS CHANGED TO ***field-expression*** GOTO ***label***, $s\rangle \ \Rightarrow\ s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![label]\!]\,]$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s >= 0$

  and

  $\mathcal{B}[\![condition\,|\,field\text{-}variable = field\text{-}expression]\!]s\ =\ \textbf{tt}$

- $[\text{FOR}_{FIRST}]$ $\langle$FOR FIRST ***pointer/field-variable*** FROM ***field-expression***$_1$ WHERE ***condition*** | ***field-variable*** IS CHANGED TO ***field-expression*** DO ***S/statement-block-name***, $s\rangle \ \Rightarrow$ $\langle$***S/statement-block-name-name***, $s\rangle$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s\ >=\ 0$

  and

  if $\mathcal{B}[\![condition\,|\,field\text{-}variable = field\text{-}expression]\!]s\ =\ \textbf{tt}$

[32]Due to space limitations, we will write only S where we mean *S/statement-block-name* in the first rule for the FOR ALL statement. For the same reason, we will say field-var and field-exp where we mean field-variable and field-expression respectively.

- [FOR$_{FIRST}$] ⟨`FOR FIRST` *pointer/field-variable* `FROM` *field-expression*$_1$
  `WHERE` *condition* | *field-variable* `IS CHANGED TO`
  field-expression `GOTO` *label*, s⟩ $\Rightarrow$
  ⟨`FOR FIRST` *pointer/field-variable* `FROM` *field-expression*$_1$ - 1
  `WHERE` *condition* | *field-variable* `IS CHANGED TO`
  field-expression `GOTO` *label*, s⟩

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ `>=` 0
  and
  if $\mathcal{B}[\![condition \,|\, field\text{-}variable = field\text{-}expression]\!]s = \textbf{ff}$

- [FOR$_{FIRST}$] ⟨`FOR FIRST` *pointer/field-variable* `FROM` *field-expression*$_1$
  `WHERE` *condition* | *field-variable* `IS CHANGED TO`
  field-expression `DO` *S/statement-block-name*, s⟩ $\Rightarrow$
  ⟨`FOR FIRST` *pointer/field-variable* `FROM` *field-expression*$_1$ - 1
  `WHERE` *condition* | *field-variable* `IS CHANGED TO`
  field-expression `DO` *S/statement-block-name*, s⟩

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ `>=` 0
  and
  if $\mathcal{B}[\![condition \,|\, field\text{-}variable = field\text{-}expression]\!]s = \textbf{ff}$

- [FOR$_{FIRST}$] ⟨`FOR FIRST` *pointer/field-variable* `FROM` *field-expression*$_1$
  `DO` *S/statement-block-name*, s⟩ $\Rightarrow$ $s[\mathcal{VSC}{+}{+}]$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ `<` 0

6.17 The Semantics for Signal Statements

As pointed out in Section 6.9, the main distinction between signals is **either** between *direct* and *buffered* **or** between *single* and *combined*. The distinction depends on whether one looks at the signals from the point of view of the execution model, or from a syntactical point of view, respectively. This means that, since we are specifying their semantics, we are only interested in their effect, not on their syntax. **But**, since the syntax is only concerned with if an answer is expected or not, not on whether the signal is direct or buffered, we need help. From Section 6.9 (and Appendix B), we recall that the information on whether the signal is direct or buffered could be found in the signal sending statement **and**

in the *Signal Description*[33] So, by defining the function

$$\mathcal{SD} : signal \rightarrow sig\text{-}property$$

where $sig\text{-}property \in \{\texttt{BUFFER, DIRECT}\}$, which inspects the Signal Description and tells whether a signal is direct or buffered, we are able to separate between these categories.

- Our general approach for the semantics of signals is to model them as *assignments* **or** '*jumps*'.

I.e., a buffered signal is put in, *or assigned to*, a job buffer, whereas a direct signal moves, *or jumps*, to the signal receiver for further execution. Fig. 6.6, which we repeat from Section 3.2.3 (Fig. 3.12), explains the mechanism of finding the receiver of a signal. From the figure, it is obvious that we can not use the same mechanism as with the jump-statements, i.e., $\mathcal{VSC} \mapsto label$, when we specify the semantics for direct signals since the execution will continue in another block (for a direct signal) or with the next statement (for a buffered signal)[34]. To solve our problem, we define the following function:

$$\mathcal{SIG} : signal \rightarrow address$$

which takes a signal as input and returns an address. This address specifies where in the receiving block the *signal receiving statement*, for "this" particular signal is found[35]. In other words, the function $\mathcal{SIG}$ will perform a "backward trace" of the linking procedure described in Fig. 6.6.

With the above functions we are able to tell whether a signal is direct or buffered and we will also be able to specify the semantics for direct signals (as will be shown).

We claimed above that a buffered signal will be modeled as an assignment to "its" job buffer. We can not, however, use the same notation as we did when we specified the semantics for assignment statements.

[33]The Signal Description is covered in Appendix B.

[34]This implies that *labels* in different blocks are disjunct sets

[35]Note from Fig. 3.3 and 6.6 that a block, or the Source Program Information (SPI, Section 2.2.1), contains one or several sub-programs and that each sub-program starts with a signal receiving statement. This means that a block contains one, or several, signal receiving statements - One for each sub-program.

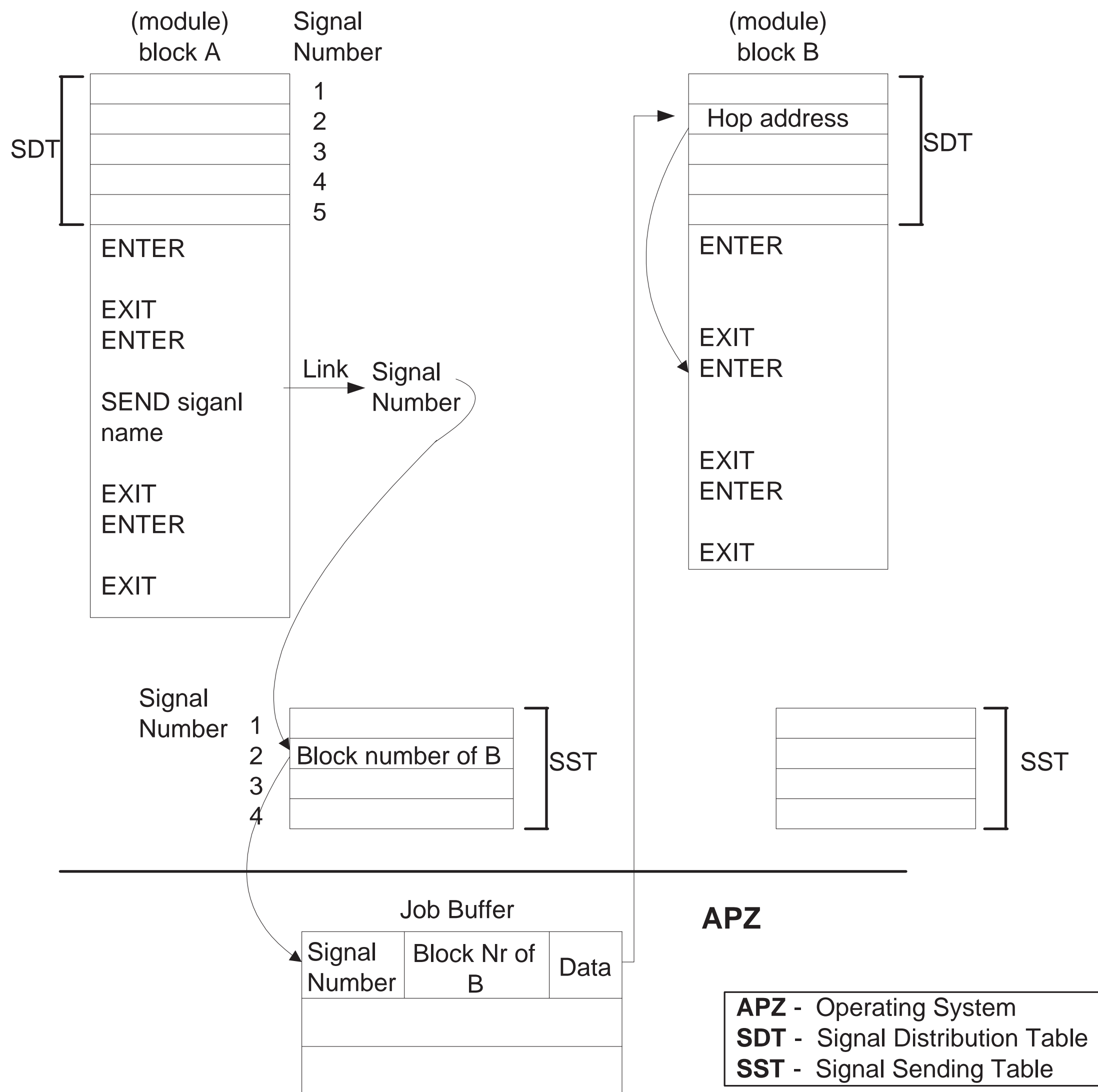


Figure 6.6: *The process of finding the receiver of a buffered signal. (Repeated from Section 3.2.3, Fig. 3.12.)* **NOTE**: *The process is similar for direct signals, except that they are not inserted in a Job Buffer.* (See also Figure 6.4 for a detailed description on the organization of the job buffers.)

The reason is that we don't change the contents of a buffer in a similar way as we change the contents of a memory location when we deal with assignments. The assignment of a buffered signal to a job buffer is more of a "insert"-operation to a queue. To capture the insert of a signal, sig, to job buffer A (for instance) we could use the notation **JBA:**sig to denote that sig is inserted at the end of job buffer A (and similar for the other buffers). But if we examine Fig. 6.4 and recapture what was said in Section 3.1.6 we realize that we must be able to separate the case where we "insert" a signal from the case where we "insert" signal **data**[36]. For this reason we will write $\mathbf{JBA}_S$**:**sig and $\mathbf{JBA}_D$**:**sig-$data$ to denote that we insert a signal and a signal-data respectively to job buffer A. And, to be able to tell which of the buffers to assign the signal to (or insert it in) we define the function:

$$\mathcal{L} : signal \rightarrow level \text{ where } level \in \{\texttt{A, B, C, D, R}\}$$

I.e., the function $\mathcal{L}$ inspects the Signal Description for a given signal and determines the buffer level for that signal.

6.17.1 Single Signals

To determine the semantics of a *single signal **sending** statement*, we will use the above defined functions $\mathcal{SD}$, $\mathcal{SDT}$ and $\mathcal{SST}$ as in the following rules. We will also take care of possible signal data and we recall, from Section 3.1.6, that single and combined signals may carry up to 25 signal data. Since "*signal data is variable values sent with a signal*" [AB98], we can use our functions $\mathcal{A}[\![\ ]\!]$ and $\mathcal{ST}[\![\ ]\!]$ to determine the semantics of the signal data. The keyword `BUFFER` will be omitted in the following rules. The reason is that this keyword "*should not be used in new design*" [AB98].

•[single-sig$_{send}$] $\langle$`SEND` *signal*, s$\rangle$ $\Rightarrow$

$$s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![signal]\!], RM \mapsto \texttt{UNDEF}]$$

$$\textbf{if } \mathcal{SD}[\![signal]\!]s \ = \ \texttt{DIRECT}$$

[36]What we actually do when a new signal is buffered is to assign the signal and its data (if any) to a new "instance" of the form described in Fig. 6.4.

- [**single-sig**$_{send}$] $\langle$`SEND` ***signal*** `WITH` ***signal-datum***$_{1-k}, s\rangle \Rightarrow$

$$s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![signal]\!],\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

if $\mathcal{SD}[\![signal]\!]s$ = `DIRECT`

Note that we have only shown the case where the signal-data is an integer value, i.e., $\mathcal{A}[\![signal\text{-}datum_k]\!]$. The case where the signal-data is a string is similar but we would use $\mathcal{ST}[\![signal\text{-}datum_k]\!]$ instead.

- [**single-sig**$_{send}$] $\langle$`SEND` ***signal***, **s**$\rangle \Rightarrow s[\mathcal{VSC}{+}{+},\ JBA_S : signal]$

if $\mathcal{SD}[\![signal]\!]s$ = `BUFFER`

and

$\mathcal{L}[\![signal]\!]s$ = `A`

- [**single-sig**$_{send}$] $\langle$`SEND` ***signal*** `WITH` ***signal-datum***$_{1-k}, s\rangle \Rightarrow$

$$s[\mathcal{VSC}{+}{+},\ JBA_S : signal,\ JBA_D : signal\text{-}datum_{1-k}]$$

if $\mathcal{SD}[\![signal]\!]s$ = `BUFFER`

and

$\mathcal{L}[\![signal]\!]s$ = `A`

The rules for the cases of $\mathcal{L}[\![signal]\!]s$ = `B, C, D` and `R` in the two rules above, are similar. We omit them in this report since there is no difference from the above rules, except that job buffer `B, C, D` or `R` are updated similar to job buffer `A` above!

To receive a single signal, we must consider the following cases:

- The signal is direct and carries no data.
- The signal is direct and carries data.
- The signal is buffered and carries no data.
- The signal is buffered and carries data.

However, the semantics for the direct and buffered signal that **does not** carry data are practically similar. In both cases, the execution will

continue after the signal receiving statement. This is, of course, also the case with a signal that **carries** data, but there are differences in how the data is retrieved as is shown in the following rules.

- [single-sig$_{rec}$] $\langle$`ENTER` ***signal*** , s$\rangle$ $\Rightarrow$ $s[\mathcal{VSC}\texttt{++},\ RM \mapsto \texttt{UNDEF}]$

  if $\mathcal{SD}[\![signal]\!]s$ = `DIRECT`

- [single-sig$_{rec}$] $\langle$`ENTER` ***signal*** `WITH` ***signal-datum***$_{1-k}$, s$\rangle$ $\Rightarrow$

$$s[\mathcal{VSC}\texttt{++},\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$

$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$

$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$

$$\vdots$$

$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$

$$]$$

  if $\mathcal{SD}[\![signal]\!]s$ = `DIRECT`

Actually, the signal data has already been put in the register memory (RM) by the *APZ Operating System*, which, of course, is the case for all kinds of transferring of signal data. We write the rule in this way just to make it explicit where the signal data is located.

In a similar way to when we inserted a signal (and its data) to a job buffer, and used the notation $\mathbf{JBA}_S$**:**sig and $\mathbf{JBA}_D$**:**sig**-**$data$, we will write sig**:**$\mathbf{JBA}_S$ and $signal\text{-}datum_{1\text{-}k}$**:**$\mathbf{JBA}_D$ to denote the contents of the job buffer (A in this case) **before** the statement is executed whereas the contents of the job buffer, after we have fetched a signal and its eventual data, is denoted $\mathbf{JBA}_D$ and $\mathbf{JBA}_S$ which indicates that the first item in the buffer has been removed.

With the same argumentation as with the *sending* of a single/buffered signal above, we omit the rules for *receiving* a single/buffered signal from any of the buffers `B - R` since there is no difference except which buffer is to be updated.

- [**single-sig**$_{rec}$] $\langle$`ENTER` ***signal*** , **s**$[signal : JBA_S]\rangle \Rightarrow$
  $s[\mathcal{VSC}\texttt{++},\ JBA_S,\ RM \mapsto \texttt{UNDEF}]$

  if $\mathcal{SD}[\![signal]\!]s = \texttt{BUFFER}$
  and
  $\mathcal{L}[\![signal]\!]s = \texttt{A}$

- [**single-sig**$_{rec}$] $\langle$`ENTER` ***signal*** `WITH` ***signal-datum***$_{1-k}$,
  $s[signal : JBA_S,\ signal\text{-}datum_{1\text{-}k} : JBA_D]\rangle \Rightarrow$

$$
\begin{aligned}
&s[\mathcal{VSC}\texttt{++},\ JBA_D,\ JBA_S,\\
&\quad (RM)PR0 \mapsto\ \mathcal{A}[\![signal\text{-}datum_1]\!],\\
&\quad (RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1\\ \texttt{UNDEF} & otherwise \end{cases}\\
&\quad (RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2\\ \texttt{UNDEF} & otherwise \end{cases}\\
&\qquad\qquad \vdots\\
&\quad (RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25\\ \texttt{UNDEF} & otherwise \end{cases}\\
&]
\end{aligned}
$$

  if $\mathcal{SD}[\![signal]\!]s = \texttt{BUFFER}$
  and
  $\mathcal{L}[\![signal]\!]s = \texttt{A}$

6.17.2 Combined Signals

As stated in the beginning of Section 6.17, the distinction that is of interest is between direct and buffered signals. The syntactical distinction between single and combined signals is of secondary interest since the combined signals are semantically equal to the single signals, **with one exception**, namely the sending of a combined *forward* signal. This is due to the fact that the control of the execution will *return* to the "caller"[37].

According to the above, all but one rule for the semantics of combined signals are given in shortened versions together with a few comments.

[37]From a semantical point of view, the combined signals can be viewed as direct signals since a combined signal can **never** be buffered 3.1.

We refer to Appendix A for the complete set of rules.

The sending of a combined forward signal is in many ways similar to the sending of a (direct) single signal. However, there are two things that we must take into consideration when we specify the following rule[38].

1. We must be able to tell what happens on ”the other side”. I.e., we must be able to trace the execution in the receiving part.

2. We must also handle the ”reply”, i.e, the jump back. (Recall from Section 3.1.3 that the execution always returns to the ”caller”.)

To be able to trace the execution in the receiving part, we will define the code to be executed as a `CodeBlock` and let $block_n$ be a meta-variable ranging over `CodeBlock`. $block_n$ is defined as

- $block_n$ `::=` S; `RET`
- $S ::= Statement;\ S \mid \varepsilon$
- $Statement$ ::= Any of the PLEX statements described in Section 6.4 - 6.10 **except** for the following statements: **RECEIVE**, **RETURN**, **ENTER** or **EXIT**.
- `RET` ::= **IF** $expression$ **RETURN** $cbsig$ **ELSE** `RET` | **RETURN** $cbsig$

I.e., we define `CodeBlock` as a sequence of PLEX statements with a *Single-Entry-Multiple-Exit* ”semantics”.

To ”fetch” a codeblock, we will use the following function

$$\mathcal{BLOCK} : address \to CodeBlock$$

which takes an address (which will be the value of our statement counter $\mathcal{VSC}$) and returns the corresponding CodeBlock. Finally, to specify the meaning of this CodeBlock, we will use the function $\mathcal{S}_{PLEX}$. The function will not be defined before Section 6.19, but we can already say that this function will determine the meaning of a sequence of PLEX statements.

[38]We will omit the ”`REFERENCE` *field-variable*” part (see Section 6.9.2) since this part does not change the meaning of the statement.

- [**combined-sig-fwd**$_{send}$]

$$\frac{\langle \texttt{RECEIVE}\ cfsig,\ s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![cfsig]\!]s,\ RM \mapsto \texttt{UNDEF}]\rangle \ \Rightarrow\ s',\quad \mathcal{S}_{PLEX}[\![block_1]\!]s' \Rightarrow s''}{\langle \texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig_1\ \texttt{IN}\ label_1\ \texttt{OR} \ldots\ cbsig_n\ \texttt{IN}\ label_n,\ s\rangle \ \Rightarrow\ s''[\mathcal{VSC} \mapsto label]}$$

$$\text{where } label = \begin{cases} \mathcal{ADR}[\![label_1]\!] & \textbf{if } RM(PR0) = cbsig_1 \\ \vdots & \\ \mathcal{ADR}[\![label_n]\!] & \textbf{if } RM(PR0) = cbsig_n \end{cases}$$

and

$$block_1 = \mathcal{BLOCK}[\![\mathcal{VSC}'{+}{+}]\!]$$

NOTE: Recall that the last statement in $block_1$ is a **RETURN** $cbsig_k$ and that the name of the return signal is found in the first register ($PR0$) of the Register Memory (RM).

The case where the combined forward signal carries signal data is equivalent except that the register memory (RM) is updated with the data. We omit this case since there is no other difference from the above rule!

- [**combined-sig-fwd**$_{rec}$] $\langle$`RECEIVE` *cfsignal* [`WITH` *signal-datum*], s$\rangle \Rightarrow$ s$[\mathcal{VSC}{+}{+},\ \ldots]$

From the receiver's point of view, it is of no interest whether the signal it receives is a single signal or a combined signal. The execution will continue with the statement that follows the receiving statement, which is shown above. Depending on if the signal does carry any data, the register memory (RM) is updated in the same way as with a single signal. (See Appendix A.)

- [**combined-sig-bwd**$_{send}$] $\langle$`RETURN` *cbsignal* [`WITH` *signal-datum*], s$\rangle \Rightarrow$ s$[\mathcal{VSC} \mapsto \mathcal{SIG}[\![cbsignal]\!],\ \ldots]$

Sending a combined backward signal, means an immediate reply to the initiating block/sender, i.e. a direct signal. Again, we refer to Appendix A for a description on how the register memory is updated (which is similar to the updating in case of a single, direct signal).

- [combined-sig-bwd$_{rec}$]
  ⟨*label*)' `RETRIEVE` *cbsignal* [`WITH` *signal-datum*], s⟩ ⇒
  s[$\mathcal{VSC}$++, ...]

The same argumentation as with the combined forward receiving statement above. After the receiving of the signal the execution continues with the statement that follows the receiving statement. Appendix A shows how the register memory is updated (in case of signal data).

6.17.3 The Semantics for Local Signals

Since the local signals are similar to direct signals (Section 3.1.4), their semantics could be regarded as similar to the semantics for the single-direct signals (Section 6.17.1). But, since they are *implemented* as a direct jump, their semantics could also be seen as similar to the semantics for unconditional jumps (Section 6.13). It is also said that "*a simple GOTO statement could replace a local signal*" [AB98]. From the same source, we also find out that local signals is not transferred via the Signal Distribution Table, which means that we can not use the same functions as in Section 6.17.1.

To solve our "problem", we will make use of the following fact: Since the signal is local, the receiving point is found in the same SPI (Section 2.2.1). This means that we can use the *signal name* as a label and "combine" the semantics for the unconditional jump and the single-direct signals! As for the previous signal statements, we will separate the case when no data is sent with the signal from the case when the signal carry one or more (up to 25) signal data.

The **main difference** between local and "global"[39] signals are that local signals preserve the temporary variables, i.e., the contents of the register memory (RM) is left unchanged if no signal data is transferred along with the signal. This will be shown in the following rules.

- [local-sig$_{send}$] ⟨`TRANSFER` *signal-name*, s⟩ ⇒
  $s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![signal\text{-}name]\!]\,]$

[39] By "global" signals we mean signals that are **not** local.

•[**local-sig**$_{send}$] ⟨TRANSFER ***signal-name*** WITH ***signal-datum***$_{1-k}, s$⟩ ⇒

$$s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![signal\text{-}name]\!],\ (RM)PR0 \mapsto signal\text{-}datum_1,$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ (RM)DR0 & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ (RM)DR23 & otherwise \end{cases}$$
$$]$$

• [**local-sig**$_{rec}$] ⟨ENTRANCE ***signal-name*, s**⟩ ⇒ $s[\mathcal{VSC}{+}{+}]$

•[**local-sig**$_{rec}$] ⟨ENTRANCE ***signal-name*** WITH ***signal-datum***$_{1-k}, s$⟩ ⇒

$$s[\mathcal{VSC}{+}{+},\ (RM)PR0 \mapsto signal\text{-}datum_1,$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ (RM)DR0 & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ (RM)DR23 & otherwise \end{cases}$$
$$]$$

6.18 The Semantics for the EXIT-statement

As explained in Section 6.10, the EXIT statement is a *deactivation* statement for a sequence of statements. The meaning of the EXIT statement is that the control is transferred back to the operating system (APZ), which selects the next signal to execute based on the contents of the job buffers. This means, in a sense, that we move our semantic approach away from the level of statements and look at what's going on in the operating system. I.e., the semantics of the EXIT statement is more a question of the semantics for the *execution model* than the language[40]. However, in contradiction to the job buffer pointers, the EXIT statement **is** part of the language, and is therefore treated here. To be able to capture what happens **after** an EXIT statement, we introduce the *partial* function

$$\mathcal{APZ} : state \hookrightarrow state$$

[40]See also Section 6.17 where the similar argumentation is made on the job buffer pointers.

which inspects the current state and assign the first "ready-job" to our statement counter, $\mathcal{VSC}$. It is defined in Table 6.5.

The function is partial function since it is undefined in the case of all job buffers being empty at the same time. The absence of a transition in this situation models the fact that the system "goes idle", i.e., it will simply wait for a RP-CP signal (see Section 3.1 and Fig. 3.2) to arrive in any of the job buffers.

So, with the introduction of $\mathcal{APZ}$ we can specify the semantics of the EXIT statement as

- [**exit**] $\langle \texttt{EXIT}, \mathbf{s} \rangle \;\Rightarrow\; s[\mathcal{VSC} \mapsto \mathcal{APZ}(s),\ RM \mapsto \texttt{UNDEF}]$

$$
\mathcal{APZ}(s) = \begin{cases}
s[\mathcal{VSC} \mapsto first(JBA)] & if\ s[JBA] \neq \emptyset^{41} \\
s[\mathcal{VSC} \mapsto first(JBB)] & if\ s[JBA] = \emptyset \quad \textbf{and} \\
 & if\ s[JBB] \neq \emptyset \\
s[\mathcal{VSC} \mapsto first(JBC)] & if\ s[JBA] = \emptyset \quad \textbf{and} \\
 & if\ s[JBB] = \emptyset \quad \textbf{and} \\
 & if\ s[JBC] \neq \emptyset \\
s[\mathcal{VSC} \mapsto first(JBD)] & if\ s[JBA] = \emptyset \quad \textbf{and} \\
 & if\ s[JBB] = \emptyset \quad \textbf{and} \\
 & if\ s[JBC] = \emptyset \quad \textbf{and} \\
 & if\ s[JBD] \neq \emptyset \\
\text{undef} & if\ s[JBA] = \emptyset \quad \textbf{and} \\
 & if\ s[JBB] = \emptyset \quad \textbf{and} \\
 & if\ s[JBC] = \emptyset \quad \textbf{and} \\
 & if\ s[JBD] = \emptyset
\end{cases}
$$

Table 6.5: *The function $\mathcal{APZ}$ which fetches the first "ready-job" with highest priority.*

[41] $JBA = \emptyset$ is the same as JBI = JBO, i.e., Job Buffer In = Job Buffer Out. (See Section 6.2.)

6.19 The Semantic Function $\mathcal{S}_{PLEX}$

In the previous sections (6.12 - 6.18) we have specified how the execution of single PLEX statements change the state of the system. In this section we will define the meaning of a *sequence of statements*. But as claimed in Section 1.2, the semantics for sequences of statements is restricted to *well-formed* constructs (i.e., well-formed sequences of statements). These sequences have a *single-entry-multiple-exit* semantics. They are entered through an **ENTER** or a **RECEIVE** statement and left with an **EXIT** or a **RETURN** statement. The entering of such a sequence, via one of the above statements, must be through a signal sending statement. I.e., every entering statement **must** be preceded by an exit statement (**EXIT** or **RETURN**) which unconditionally breaks the ”normal” sequential execution order. And, since we say that these constructs have a single-entry semantics, we do not allow a second entering statement before an exit statement!

To specify the semantics for sequences of statements in a proper way, there are two important things that we must take into consideration:

1. We must capture the possibility of a ”non-sequential” execution order. I.e., the possible occurrence of a `GOTO` statement must be handled.

2. An `EXIT` statement aborts the execution and none of the statements that may follow in the source code file will be executed.

To specify the meaning of a sequence of statements, we first denote the syntactic category of *sequences of statements*, terminated by an **EXIT** statement or an unconditional **GOTO** statement, as `Code` and let c_n be a meta-variable ranging over `Code`. c_n is then defined as

- c_n ::= S; `OUT`
- S ::= $Statement$; S | ε
- $Statement$ ::= Any of the PLEX statements described in Section 6.4 - 6.10 **except** for the **EXIT** statement (Section 6.18) and the unconditional jump, **GOTO**, (Section 6.5).
- `OUT` ::= **EXIT** | **GOTO** $label$

Then, we define the function

$$\mathcal{CODE} : address \rightarrow code$$

to be the function that takes an address (which will be the value of our statement counter, $\mathcal{VSC}$) and returns the code sequence that starts at the given address.

The meaning of a sequence of statements is now given by the function $\mathcal{S}_{PLEX}$:

$$\mathcal{S}_{PLEX}[\![S;\ c_1]\!]s = \begin{cases} s' & \textbf{if } \langle S,\ s\rangle \Rightarrow s' \textbf{ and } c_1 = \varepsilon \\ s' & \textbf{if } \langle S,\ s\rangle \Rightarrow s' \textbf{ and } S = \texttt{EXIT} \\ \mathcal{S}_{PLEX}[\![c_1]\!]s' & \textbf{if } \langle S,\ s\rangle \Rightarrow s' \textbf{ and } \mathcal{VSC}' = \mathcal{VSC}\texttt{++} \\ \mathcal{S}_{PLEX}[\![c_2]\!]s' & \textbf{if } \langle S,\ s\rangle \Rightarrow s',\ \mathcal{VSC}' \neq \mathcal{VSC}\texttt{++}, \\ & \quad \textbf{and } c_2 = \mathcal{CODE}[\![\mathcal{VSC}']\!] \end{cases}$$

where s is the initial state.

Note, the last case expresses the fact that the sequential execution order is ”broken” (for example by a `GOTO` statement) and the execution is ”transferred” somewhere different from the following statement.

So, with the definition of $\mathcal{S}_{PLEX}$ we have finally specified the semantics for individual PLEX statements as well as for sequences of statements. And by this, we have also specified the semantics of an entire job (Section 3.2.1) since $\mathcal{S}_{PLEX}$ determines the meaning of an arbitrary sequence of statements which could very well start with an `ENTER` statement and end with an `EXIT` statement (which constitutes a job).

Chapter 7

Summary

This report presents a structural operational semantics for the language PLEX, a pseudo-parallel and event-based real-time language developed by *Ericsson* and used to program telephony systems, especially central parts of the AXE switching system (from Ericsson). We have presented a formal description of fundamental parts of the language, which is considered to be jumps and signal sending statements.

By means of the semantics presented in this report, a formal basis for further investigations and comparisons with other languages is provided since the *meaning*, i.e. the semantics, of the language now is explicit. The semantics presented could also be seen as a reference manual when the language is implemented for new hardware platforms.

This report should also be seen in a further perspective, where the aim is to extend and modify the language with a possibility to run in a multi-processor environment, as described in Section 1.1. With a semantic description of fundamental parts of the language, an important step towards executing PLEX in a multi-processor environment has been taken.

Acknowledgements

This report was written mainly between spring and autumn 2002 at *Mälardalen University*, Västerås, Sweden. The work is published within the research co-operation between *Ericsson AB* and *Mälardalen Real-Time Research Center*.

First of all, the author would like to thank ***Björn Lisper*** at Mälardalen University, and ***Janet Wennersten*** at Ericsson AB, for supervising this work.

Secondly, a number of people at Mälardalen University, have helped me in different ways: I have had many interesting discussions with ***Peter Funk*** and ***Bo Lindell***. ***Lars Bruce*** and ***Jan Gustafsson*** have both proof read parts of the material. ***Markus Bohlin*** and ***Jan Carlson*** have helped me out when my LaTeX and Emacs skills were insufficient.

Then comes a number of people at Ericsson AB: ***Per-Åke Ek***, ***Anton Massoud***, ***Ingvar Nilsson*** and ***Lars-Erik Wiman*** have all helped me with material and/or discussions.

I would also like to thank ***Per Burman***, ***Anders R. Larsson*** and ***Anders Skelander*** at Ericsson AB, as well as ***Peter Funk*** (again) at Mälardalen University, who all have been eager to keep the research co-operation between Ericsson AB and Mälardalen University ”up and running”.

Finally, this work has been founded by ***Ericsson AB***, ***Mälardalen Real-Time Research Center*** and ***the KK-foundation***. Thank you.

Bibliography

[AB86] Ericsson Telecom AB. *PLEX Reference Manual*, 1986.

[AB98] Ericsson Telecom AB. *PLEX-C 1 Course book*, 1998.

[AB02] Ericsson Telecom AB. *PLEX-C Language Description*, 2002.

[AE00] J. Axelsson and J. Erikson. SAPP, Theories and Tools for Execution Time Estimation for Soft Real Time (Communication) Systems. Master's thesis, Mälardalen University, 2000.

[AGG99] A. Arnstrom, C. Grosz, and A. Guillemot. GRETA: a tool concept for validation and verification of signal based systems (e.g. written in PLEX). Master's thesis, Mälardalen University, 1999.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[EGG+01] R. Eschbach, U. Glässer, R. Gotzhein, M. von Löwis, and A. Prinz. Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 11(7):1025–1050, 2001.

[EL02] J. Erikson and B. Lindell. The Execution Model of the APZ/PLEX - An Informal Description. Technical report, Mälardalen University, 2002.

[Her99] D. Herzberg. UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain. In *Proceedings of the Second International Conference on UML (UML'99),LNCS 1723, Springer*, 1999.

[IT82] ITU-T. *CHILL: Formal Definition*, 1982. International Telecommunication Union, Volume 1, Part 1, 2, 3.

[IT99] ITU-T. *CHILL: The ITU-T programming Language*, 11 1999. International Telecommunication Union, Geneva, (Recommendation Z.200).

[IT00] ITU-T. *SDL Formal Semantics Definition*, 2000. ITU-T Recomendation Z.100 Annex F.

[KP96] Al Kelley and Ira Pohl. *C By Dissection - The Essentials of C Programming*. Addison-Wesley, 1996.

[Lin03] B. Lindell. Analysis of reentrancy and problems of data interference in the parallel execution of a multi processor AXE-APZ system. Master's thesis, Mälardalen University, 2003.

[Lis98] B. Lisper. *Semantical Tidbits*, 1998.

[Mos01] P. D. Mosses. The Varieties of Programming Language Semantics (and Their Uses). In *Lecture Notes in Computer Science, Vol. 2244*, pages 165 – 190, 2001.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[Win00] Jürgen F. H. Winkler. CHILL 2000. *Telektronikk*, 96(4):70–77, 2000.

Appendix A

The Semantics for PLEX

In this section, we have collected the semantics for the individual PLEX statements that was given (sometimes in shortened versions) in Chapter 6.

The Semantics for Assignment Statements

- [ass_{field}] $\langle x := a, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++},\ (DS)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!]$ = DS
- [ass_{field}] $\langle x := a, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++},\ (RM)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!] = \varepsilon$
- [ass_{sym}] $\langle x := a, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++},\ (DS)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!]$ = DS
- [ass_{sym}] $\langle x := a, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++},\ (RM)x \mapsto \mathcal{A}[\![a]\!]s]$ if $\mathcal{P}[\![x]\!] = \varepsilon$
- [ass_{string}] $\langle x := c, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++},\ (DS)x \mapsto \mathcal{ST}[\![c]\!]s]$ if $\mathcal{P}[\![x]\!]$ = DS
- [ass_{string}] $\langle x := c, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++},\ (RM)x \mapsto \mathcal{ST}[\![c]\!]s]$ if $\mathcal{P}[\![x]\!] = \varepsilon$

The Semantics for Jump Statements

- [jump_{uncond}] $\langle \texttt{GOTO}\ label, s\rangle \;\Rightarrow\; s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![label]\!]\ ]$
- [jump_{cond}] $\langle \texttt{IF}\ condition\ \texttt{GOTO}\ label, s\rangle \;\Rightarrow\; s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![label]\!]\ ]$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{tt}$
- [jump_{cond}] $\langle \texttt{IF}\ condition\ \texttt{GOTO}\ label, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\text{++}]$

  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{ff}$

The Semantics for Conditional Statements

- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1, s\rangle \;\Rightarrow\; \langle S_1,\ s\rangle$
  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{tt}$
- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1, s\rangle \;\Rightarrow\; s[\mathcal{VSC}\texttt{++}]$
  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{ff}$
- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2, s\rangle \;\Rightarrow\; \langle S_1,\ s\rangle$
  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{tt}$
- [$\mathbf{cond}_{if}$] $\langle \texttt{IF}\ condition\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2, s\rangle \;\Rightarrow\; \langle S_2,\ s\rangle$
  $if\ \mathcal{B}[\![condition]\!]s = \mathbf{ff}$

The Semantics for Selection Statements

- [**select**] $\langle \texttt{CASE}$ ***expression*** $\texttt{IS WHEN}$ ***choice*** $\texttt{DO}\ S_1$
  $\texttt{OTHERWISE DO}\ S_n, s\rangle \Rightarrow \langle S_1, s\rangle$
  $if\ \mathcal{B}[\![expression = choice]\!]s = \mathbf{tt}$
- [**select**] $\langle \texttt{CASE}$ ***expression*** $\texttt{IS WHEN}$ ***choice*** $\texttt{DO}\ S_1$
  $\texttt{OTHERWISE DO}\ S_n, s\rangle \Rightarrow \langle S_n, s\rangle$
  $if\ \mathcal{B}[\![expression = choice]\!]s = \mathbf{ff}$

The Semantics for Iteration Statements

- [ON_{Up}]

$$\frac{\langle S,\ s\rangle \Rightarrow s',\ \langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1{+}1\ \texttt{UPTO}\ exp_2\ \texttt{DO}\ S,\ s'\rangle \Rightarrow s''}{\langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1\ \texttt{UPTO}\ exp_2\ \texttt{DO}\ S,\ s\rangle \Rightarrow s''}$$

$$if\ \mathcal{A}[\![exp_1]\!]s\ <\ \mathcal{A}[\![exp_2]\!]s$$

- [ON_{Up}] $\langle \texttt{ON}$ ***pointer/variable*** $\texttt{FROM}$ '***expression***$_1$' $\texttt{UPTO}$
  '***expression***$_2$' $\texttt{DO}\ S,\ s\rangle \Rightarrow\ s[\mathcal{VSC}\texttt{++}]$
  $if\ \mathcal{A}[\![expression_1]\!]s\ >\ \mathcal{A}[\![expression_2]\!]s$
- [ON_{Down}]

$$\frac{\langle S,\ s\rangle \Rightarrow s',\ \langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1{-}1\ \texttt{DOWNTO}\ exp_2\ \texttt{DO}\ S,\ s'\rangle \Rightarrow s''}{\langle \texttt{ON}\ point/var\ \texttt{FROM}\ exp_1\ \texttt{DOWNTO}\ exp_2\ \texttt{DO}\ S,\ s\rangle \Rightarrow s''}$$

$$if\ \mathcal{A}[\![exp_1]\!]s\ >\ \mathcal{A}[\![exp_2]\!]s$$

- [ON_{Down}] ⟨`ON` ***pointer/variable*** `FROM` '***expression***$_1$' `DOWNTO` '***expression***$_2$' `DO` $S,\ s\rangle \Rightarrow\ s[\mathcal{VSC}{+}{+}]$

  $if\ \mathcal{A}[\![expression_1]\!]s\ <\ \mathcal{A}[\![expression_2]\!]s$

- [FOR_{ALL}]

$$\frac{\langle S,\ s\rangle\ \Rightarrow\ s',\quad \langle \texttt{FOR ALL}\ pointer/field\texttt{-}var\ \texttt{FROM}\ field\texttt{-}exp_1\ \texttt{- 1 DO}\ S,\ s'\rangle \Rightarrow s''}{\langle \texttt{FOR ALL}\ pointer/field\texttt{-}var\ \texttt{FROM}\ field\texttt{-}exp_1\ \texttt{DO}\ S,\ s\rangle \Rightarrow s''}$$

  $if\ \mathcal{A}[\![field - exp_1]\!]s\ >\ 0$

- [FOR_{ALL}] ⟨`FOR ALL` ***pointer/field-variable*** `FROM` ***field-expression***$_1$ `DO` ***S/statement-block-name***, $s\rangle\ \Rightarrow\ s[\mathcal{VSC}{+}{+}]$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s\ <\ 0$

- [FOR_{FIRST}] ⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$ `WHERE` ***condition*** | ***field-variable*** `IS CHANGED TO` ***field-expression*** `GOTO` ***label***, $s\rangle\ \Rightarrow\ s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![label]\!]\ ]$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ >= 0

  and

  $\mathcal{B}[\![condition\,|\,field\text{-}variable = field\text{-}expression]\!]s$ = **tt**

- [FOR_{FIRST}] ⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$ `WHERE` ***condition*** | ***field-variable*** `IS CHANGED TO` ***field-expression*** `DO` ***S/statement-block-name***, $s\rangle\ \Rightarrow$ ⟨***S/statement-block-name-name***, $s\rangle$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ >= 0

  and

  if $\mathcal{B}[\![condition\,|\,field\text{-}variable = field\text{-}expression]\!]s$ = **tt**

- [FOR_{FIRST}] ⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$ `WHERE` ***condition*** | ***field-variable*** `IS CHANGED TO` ***field-expression*** `GOTO` ***label***, $s\rangle\ \Rightarrow$

  ⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$ - 1 `WHERE` ***condition*** | ***field-variable*** `IS CHANGED TO` ***field-expression*** `GOTO` ***label***, $s\rangle$

  if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ >= 0

  and

  if $\mathcal{B}[\![condition\,|\,field\text{-}variable = field\text{-}expression]\!]s$ = **ff**

• [FOR$_{FIRST}$] ⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$
`WHERE` ***condition*** | ***field-variable*** `IS CHANGED TO`
field-expression `DO` ***S/statement-block-name***, $s\rangle \Rightarrow$
⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$ - 1
`WHERE` ***condition*** | ***field-variable*** `IS CHANGED TO`
field-expression `DO` ***S/statement-block-name***, $s\rangle$

if $\mathcal{A}[\![field\text{-}expression_1]\!]s$ `>=` 0

and

if $\mathcal{B}[\![condition \,|\, field\text{-}variable = field\text{-}expression]\!]s = \mathbf{ff}$

• [FOR$_{FIRST}$] ⟨`FOR FIRST` ***pointer/field-variable*** `FROM` ***field-expression***$_1$
`DO` ***S/statement-block-name***, $s\rangle \Rightarrow s[\mathcal{VSC}{+}{+}]$

if $\mathcal{A}[\![field\text{-}expression_1]\!]s < 0$

The Semantics for Signal Statements

•[**single-sig**$_{send}$] ⟨`SEND` ***signal***, **s**⟩ $\Rightarrow$
$s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![signal]\!], RM \mapsto$ `UNDEF`$]$

if $\mathcal{SD}[\![signal]\!]s$ = `DIRECT`

•[**single-sig**$_{send}$] ⟨`SEND` ***signal*** `WITH` ***signal-datum***$_{1-k}, s\rangle \Rightarrow$

$$s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![signal]\!],\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

if $\mathcal{SD}[\![signal]\!]s$ = `DIRECT`

• [**single-sig**$_{send}$] ⟨`SEND` ***signal***, **s**⟩ $\Rightarrow s[\mathcal{VSC}{+}{+},\ JBA_S : signal]$

if $\mathcal{SD}[\![signal]\!]s$ = `BUFFER`

and

$\mathcal{L}[\![signal]\!]s$ = `A`

- [**single-sig**$_{send}$] $\langle$`SEND` ***signal*** `WITH` ***signal-datum***$_{1-k}, s\rangle \Rightarrow$
$s[\mathcal{VSC}\texttt{++},\ JBA_S : signal,\ JBA_D : signal\text{-}datum_{1-k}]$

$$\textbf{if } \mathcal{SD}[\![signal]\!]s = \texttt{BUFFER} \quad \textbf{and} \quad \mathcal{L}[\![signal]\!]s = \texttt{A}$$

- [**single-sig**$_{rec}$] $\langle$`ENTER` ***signal*** , **s**$\rangle \Rightarrow s[\mathcal{VSC}\texttt{++},\ RM \mapsto \texttt{UNDEF}]$

$$\textbf{if } \mathcal{SD}[\![signal]\!]s = \texttt{DIRECT}$$

- [**single-sig**$_{rec}$] $\langle$`ENTER` ***signal*** `WITH` ***signal-datum***$_{1-k}, s\rangle \Rightarrow$

$$s[\mathcal{VSC}\texttt{++},\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

$$\textbf{if } \mathcal{SD}[\![signal]\!]s = \texttt{DIRECT}$$

- [**single-sig**$_{rec}$] $\langle$`ENTER` ***signal*** , **s**$[signal : JBA_S]\rangle \Rightarrow$
$s[\mathcal{VSC}\texttt{++},\ JBA_S,\ RM \mapsto \texttt{UNDEF}]$

$$\textbf{if } \mathcal{SD}[\![signal]\!]s = \texttt{BUFFER} \quad \textbf{and} \quad \mathcal{L}[\![signal]\!]s = \texttt{A}$$

- [**single-sig**$_{rec}$] $\langle$`ENTER` ***signal*** `WITH` ***signal-datum***$_{1-k}$,

$$s[signal : JBA_S,\ signal\text{-}datum_{1\text{-}k} : JBA_D]\rangle \ \Rightarrow$$
$$s[\mathcal{VSC}\texttt{++},\ JBA_D,\ JBA_S,$$
$$(RM)PR0 \mapsto\ \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

if $\mathcal{SD}[\![signal]\!]s = \texttt{BUFFER}$

and

$\mathcal{L}[\![signal]\!]s \ = \ \texttt{A}$

- [**combined-sig-fwd**$_{send}$]

$$\frac{\langle \texttt{RECEIVE}\ cfsig,\ s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![cfsig]\!]s,\ RM \mapsto \texttt{UNDEF}]\rangle \ \Rightarrow\ s',\quad \mathcal{S}_{PLEX}[\![block_1]\!]s' \Rightarrow s''}{\langle \texttt{SEND}\ cfsig\ \texttt{WAIT FOR}\ cbsig_1\ \texttt{IN}\ label_1\ \texttt{OR} \ldots\ cbsig_n\ \texttt{IN}\ label_n,\ s\rangle}$$
$$\Rightarrow\ s''[\mathcal{VSC} \mapsto label]$$

where ***label*** $= \begin{cases} \mathcal{ADR}[\![label_1]\!] & \textbf{if } RM(PR0) = cbsig_1 \\ \vdots & \\ \mathcal{ADR}[\![label_n]\!] & \textbf{if } RM(PR0) = cbsig_n \end{cases}$

and

$block_1 = \mathcal{BLOCK}[\![\mathcal{VSC}'\texttt{++}]\!]$

- [**combined-sig-fwd**$_{rec}$] $\langle$`RECEIVE` ***cfsignal***, **s**$\rangle \Rightarrow$
  s$[\mathcal{VSC}\texttt{++},\ RM \mapsto \texttt{UNDEF}]$

- [**combined-sig-fwd**$_{rec}$] $\langle$`RECEIVE` ***cfsignal*** `WITH` ***signal-datum***$_{1-k},\ s\rangle\ \Rightarrow$

$$s[\mathcal{VSC}\texttt{++},\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

- [**combined-sig-bwd**$_{send}$] $\langle$`RETURN`***cbsignal***$,\ s\rangle \Rightarrow$

$$\mathbf{s}[\mathcal{VSC} \mapsto \mathcal{SIG}[\![cbsignal]\!], RM \mapsto \texttt{UNDEF}]$$

- [**combined-sig-bwd**$_{send}$] $\langle$`RETURN` ***cbsignal*** `WITH` ***signal-datum***$_{1-k},\ s\rangle\ \Rightarrow$

$$s[\mathcal{VSC} \mapsto \mathcal{SIG}[\![cbsignal]\!],\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

- [**combined-sig-bwd**$_{rec}$]

$$\langle label')'\ \texttt{RETRIEVE}\ cbsignal,\ s\rangle \Rightarrow s[\mathcal{VSC}\texttt{++},\ RM \mapsto \texttt{UNDEF}]$$

- [**combined-sig-bwd**$_{rec}$]

$$\langle label')'\ \texttt{RETRIEVE}\textit{\textbf{cbsignal}}[\texttt{WITH}\ signal - datum], s\rangle \Rightarrow$$
$$\mathbf{s}[\mathcal{VSC}\texttt{++},\ (RM)PR0 \mapsto \mathcal{A}[\![signal\text{-}datum_1]\!],$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$(RM)DR1 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_3]\!] & \textbf{if } k > 2 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ \texttt{UNDEF} & otherwise \end{cases}$$
$$]$$

- [**local-sig**$_{send}$] $\langle \texttt{TRANSFER}\ signal\text{-}name,\ s\rangle \ \Rightarrow$
$$s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![signal\text{-}name]\!]\ ]$$

- [**local-sig**$_{send}$] $\langle \texttt{TRANSFER}\ signal\text{-}name\ \texttt{WITH}\ signal\text{-}datum_{1-k},\ s\rangle \ \Rightarrow$
$$s[\mathcal{VSC} \mapsto \mathcal{ADR}[\![signal\text{-}name]\!],\ \ (RM)PR0 \mapsto\ signal\text{-}datum_1,$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ (RM)DR0 & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ (RM)DR23 & otherwise \end{cases}$$
$$]$$

- [**local-sig**$_{rec}$] $\langle \texttt{ENTRANCE}\ signal\text{-}name,\ s\rangle \ \Rightarrow\ s[\mathcal{VSC}\texttt{++}]$

- [**local-sig**$_{rec}$] $\langle \texttt{ENTRANCE}\ signal\text{-}name\ \texttt{WITH}\ signal\text{-}datum_{1-k}, s\rangle \ \Rightarrow$
$$s[\mathcal{VSC}\texttt{++},\ \ (RM)PR0 \mapsto\ signal\text{-}datum_1,$$
$$(RM)DR0 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_2]\!] & \textbf{if } k > 1 \\ (RM)DR0 & otherwise \end{cases}$$
$$\vdots$$
$$(RM)DR23 \mapsto \begin{cases} \mathcal{A}[\![signal\text{-}datum_{25}]\!] & \textbf{if } k = 25 \\ (RM)DR23 & otherwise \end{cases}$$
$$]$$

The Semantics for the EXIT Statement

- [**exit**] $\langle \texttt{EXIT},\ s\rangle \ \Rightarrow\ s[\mathcal{VSC} \mapsto \mathcal{APZ}(s),\ RM \mapsto \texttt{UNDEF}]$

Appendix B

The Signal Description

There are two main documents for signals: The *Signal Survey*, which is a listing of all the signals sent and received in a unit, and the *Signal Description*, which will be studied in this section. These documents are compiled together with the *Source Program Information*[1], SPI.

The Signal Description, SD, is the document that *defines* a signal. The type of the signal, as well as the priority level of the signal is specified in this document. There is *one SD for every signal* and all SD's are stored in special libraries. We will study how the SD "interact" with the SPI during the code generation phase. But as a first attempt to capture the contents of the SD, it could be seen as similar to the `h-file` in `C` that externally defines a function (among other things).
The SD includes the following items:

- **Name of the signal** - Every signal has a name that, perhaps, captures its functionality.
- **Signal number** - For internal documentation.
- **Function** - Used to make comments about functionality.
- **Signal type** - The signal type indicates whether the signal is *Single* or *Combined*. There are three possible type specifications:
  - **Type 1**: Single signal

[1]The Source Program Information is the "source code file", i.e., the document that we normally call a program. See Section 2.2.1 for further details.

  - **Type 2**: Combined forward
  - **Type 3**: Combined backward

  If the signal is *Multiple*, this is indicated by adding the keyword **MULTIPLE** to the signal type, like in: `TYPE IS 1 MULTIPLE`
  A *Unique* signal has no indication at this point! (A signal is unique ”by default” if nothing else is stated.) If the signal is local, the keyword **LOCAL** is added to the signal type.

- *Possible* **return signal** - For internal documentation.

- *Possible* **sending block** - For internal documentation.

- *Possible* **receiving block** - For internal documentation.

- **Buffer level** - The priority level! In Section 3.2, it is described how every signal is assigned a priority level, and how signals are stored in different *job buffers* (**in case of a buffered signal**). The buffer level states the priority level of the corresponding job and also in which job buffer the signal will be buffered (if it is to be buffered, i.e.).
  NOTE: The buffer level can have the following combinations:

  - **NO BUFFER**: The signal is direct. (A combined signal is **always** direct, see Section 3.1.3.)
  - **LEVEL A/B/C/D BUFFER**: The signal is buffered and uses the job buffer specified. This combination *overrides* a possible use of the `HURRY` option in the signal sending statement (see below).
  - **LEVEL A/B/C/D**: The signal is buffered and uses the job buffer specified, ***unless*** the keyword `HURRY` is used in the signal sending statement, which indicates that the signal is direct.

- **Signal data** - Specification of the ”arguments” (i.e., the data) that the signal is carrying.

- **ID sector** - For internal documentation.